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

Diszkrét dinamikus rendszerek viselkedésének felderítése ellenpélda-alapú absztrakció finomítás (CEGAR) segítségével

TDK-dolgozat

Készítette:
Hajdu Ákos
Mártonka Zoltán

Konzulensek:
dr. Bartha Tamás
Vörös András

2012

Tartalomjegyzék

1. fejezet

Bevezető

A Petri-hálók többek között az aszinkron, elosztott, párhuzamos és nem-determinisztikus rendszerek elterjedt grafikus és matematikai modellező eszközei. A Petri-háló modellek viselkedését az elérhető állapotok és az állapotátmenetek halmaza (az állapottér) adja meg. A modell fontos tulajdonsága, hogy akár végtelen nagy állapottereket is képes kezelni és vizsgálni.

Az egyik legfontosabb Petri-hálókkal kapcsolatos vizsgálat az ún. elérhetőségi analízis (lásd 2.2. fejezet) is az állapottérhez kapcsolódik: azt vizsgálja, hogy egy adott állapot elérhető-e engedélyezett állapotátmenetek sorozatával egy adott kezdőállapotból kiindulva. Ezen eldönthetőségi probléma a matematika nehéz problémái közé esik, a komplexitására (lásd 2.2.4. fejezet) még nem tudtak felső korlátot adni, az irodalom alapján legalább az EXSPACE-nehéz komplexitási osztályba esik.

Az ellenpélda-alapú absztrakció finomítás (lásd 3. fejezet) (angolul Counter Example Guided Abstracion Refinement, röviden CEGAR) gyakran használt megközelítés végtelen állapotterű rendszerek vizsgálatára. Lényege, hogy az állapottér egy absztrakcióján vizsgáljuk az elérhetőségi kérdést, és ha kell, a konkrét állapotok bejárása során nyert információ segítségével finomítjuk az absztrakciót. A 2011. évi Petri-háló modellellenőrző versenyen [4] több kategóriát is megnyert egy CEGAR alapú, elérhetőséget vizsgáló algoritmus. Ez az algoritmus a Petri-hálók állapotegyenletét használja absztrakcióként, és ezt finomítja mindaddig, amíg talál egy, az elérhetőséget igazoló állapotátmeneti útvonalat, vagy bizonyítja, hogy az állapot nem elérhető. Munkánk során ezt az algoritmust vizsgáltuk és fejlesztettük tovább.

Dolgozatunkban elméleti eredményként bemutatjuk, hogy az algoritmus bizonyos esetekben nem tudja megállapítani az elérhetőséget (lásd 3.3. fejezet), és megoldást javasolunk, amely felhasználásával a problémák nagyobb köre esetén jut az algoritmus eredményre. Bemutatunk olyan optimalizációkat, amelyek kiküszöbölik az eredeti algoritmus bizonyos esetekben jelentkező hiányosságait, és hatékonyan vágják a keresési teret. Ezen túlmenően kiterjesztettük az algoritmust, hogy a problémák bővebb osztályát is kezelni tudjuk:

- alkalmassá tettük az algoritmust, hogy predikátumokkal definiált elérhetőségi vagy fedési problémákat is tudjon hatékonyan vizsgálni (lásd 3.3.1. fejezet),
- bővítettük az algoritmust, hogy a tiltó éleket tartalmazó Petri-hálók vizsgálatára is alkalmas legyen (lásd 4. fejezet).

Ez utóbbi kiterjesztés jelentőségét az adja, hogy a tiltó élek magasabb, a Turing gépekkel azonos szintre emelik a Petri-hálók kifejező erejét. Ugyanakkor elméletileg is bizonyított tény [6], hogy tiltó élek használata esetén nem várható el teljesség az algoritmustól.

Dolgozatunkban bemutatjuk az eredeti algoritmus gyakorlati implementációjával szerzett tapasztalatainkat, a felfedezett hiányosságokat, és az ezek kiküszöbölésére és az algoritmus teljesítményének növelésére kidolgozott megoldásainkat (lásd 5. fejezet). A továbbfejlesztett algoritmus hatékonyságát mérési eredményekkel (lásd 6. fejezet) is alátámasztjuk.

2. fejezet

Háttérismeretek

Ebben a fejezetben azokat az alapismereteket és definíciókat ismertetjük, amelyek a dolgozat további részeinek megértéséhez szükségesek. Bemutatjuk az egyszerű Petri-hálókat (lásd 2.1. fejezet), kiegészítésüket tiltó élekkel (2.1.3. fejezet) és az elérhetőségi problémát (2.2. fejezet). Ismertetjük továbbá a lineáris programozás (2.3. fejezet) és az egészértékű lineáris programozás (2.3.1. fejezet) általunk használt részeit.

2.1. Petri-hálók

A Petri-hálók diszkrét dinamikus rendszerek modellezésének és analízisének elterjedt eszközei. Előnyük, hogy matematikai reprezentációjuk mellett bizonyos méretekig grafikusan is jól áttekinthetők és vizsgálhatók. Fő alkalmazási területük a konkurens, aszinkron, elosztott rendszerek modellezése [15]. Kifejezőerejük növelésének érdekében számos kiegészítésük jött létre, ezek közül mi a tiltó élekkel bővített hálókat fogjuk használni.

2.1.1. Egyszerű Petri-hálók

Az egyszerű Petri-háló egy irányított, súlyozott páros gráf. Az egyik csúcsosztály elemeit *hely*eknek (Place, P), a másik csúcsosztály elemeit pedig *tranzíció*knak (Transition, T) nevezzük. A gráfban minden irányított él egy helyet és egy tranzíciót köt össze. Az élekhez rendelt pozitív egész számot az él súlyának nevezzük. A Petri-háló állapotát egy olyan függvénnyel írhatjuk le, amely minden helyhez egy nemnegatív egész számot rendel. Ez a háló *tokeneloszlás*a, a helyekhez rendelt számok pedig a helyeken lévő *token*ek számát reprezentálják[1]. Formálisan tehát a Petri-háló egy olyan $PN = (P, T, E, W)$ struktúra [13], ahol:

$P = \{p_1, p_2, \ldots, p_k\}$ a helyek véges halmaza,
$T = \{t_1, t_2, \ldots, t_n\}$ a tranzíciók véges halmaza,
$E \subseteq (P \times T) \cup (T \times P)$ az élek halmaza,
$W : E \to \mathbb{Z}^+$ a súlyfüggvény.

A Petri-háló grafikus reprezentációjában a helyeket körökkel, a tranzíciókat téglalapokkal, az éleket nyilakkal jelöljük. Ha egy él súlya nem 1, akkor az él mellett azt feltüntetjük. A tokeneloszlást a körökbe rajzolt fekete pontok, vagy nagyobb tokenszám esetén a körökbe írt számok jelölik.

Jelölje $\bullet t$ a $t \in T$ tranzíció bemeneti helyeit, $t\bullet$ a kimeneti helyeit, valamint $\bullet p$ a $p \in P$ hely bemeneti tranzícióit, $p\bullet$ a kimeneti tranzícióit, azaz formálisan:

[1] Az állapotokat gyakran egy sorvektorral jelöljük, ahol a sorvektor i. eleme az i. hely tokenjeinek száma. Például az $m = (4, 2, 5)$ jelentése, hogy p_0 helyen 4, p_1 helyen 2, p_2 helyen pedig 5 token van, amennyiben a helyeket 0-tól számozzuk.

$\bullet t = \{p \mid (p,t) \in E\}$
$t\bullet = \{p \mid (t,p) \in E\}$
$\bullet p = \{t \mid (t,p) \in E\}$
$p\bullet = \{t \mid (p,t) \in E\}$

Tranzíciók tüzelése

A Petri-hálók dinamikus működését a tranzíciók *tüzelés*ei határozzák meg. A szabályok a következők:

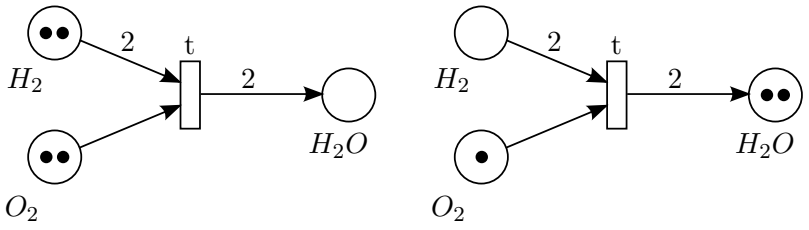
- Egy $t \in T$ tranzíció *engedélyezett*, ha minden $p \in \bullet t$ bemeneti helyén legalább $w^-(p,t)$ darab token van az adott m tokeneloszlás mellett, ahol $w^-(p,t)$ a (p,t) él súlya. Az engedélyezettséget $m[t\rangle$ -vel jelöljük.

- Egy engedélyezett tranzíció eltüzelhet, de nem kell feltétlenül tüzelnie. Ha több tranzíció is engedélyezett, akkor bármelyik eltüzelhet, ezáltal a működés nemdeterminisztikus.

- Amennyiben egy engedélyezett $t \in T$ tranzíció eltüzel, minden $p \in \bullet t$ bemeneti helyről $w^-(p,t)$ tokent vesz el, és minden $p \in t\bullet$ kimeneti helyre $w^+(p,t)$ tokent helyez el, ahol $w^+(p,t)$ a (t,p) él súlya.

Ha a kapott új állapot m', akkor a tüzelést $m[t\rangle m'$ -vel jelöljük [13]. Egy $\sigma \in T^n$ vektort *tüzelési sorozat*nak nevezünk. Amennyiben léteznek $m, m_1, m_2, \ldots, m_{n-1}, m'$ állapotok úgy, hogy $m[\sigma(0)\rangle m_1[\sigma(1)\rangle m_2 ... m_{n-1}[\sigma(n)\rangle m'$, akkor σ-t végrehajthatónak (vagy realizálhatónak) nevezzük és röviden $m[\sigma\rangle m'$ -vel jelöljük. Egy tüzelési sorozat *Parikh kép*e a $\wp(\sigma) : T \to \mathbb{N}$ vektor, ahol $\wp(\sigma)(t)$ a $t \in T$ tranzíció előfordulásainak száma a tüzelési sorozatban. Ezt a vektort tüzelési szám vektornak is nevezik.

1. példa. *A 2.1. ábrán egy Petri-háló látható, amely egy egyszerű kémiai folyamatot modellez [13]. A hálóban egy tranzíció (t) és három hely (H_2, O_2, H_2O) található. A 2.1(a) ábrán a t tranzíció engedélyezett. A t tranzíció tüzelésével a 2.1(a) ábrán látható állapotból a 2.1(b). ábrán látható állapotba kerül a háló. Ebben az állapotban nincs engedélyezett tranzíció.*



(a) Kezdő tokeneloszlás (b) Tokeneloszlás t tüzelése után

2.1. ábra. Példa Petri-háló: hidrogén oxidációja

Elérhetőség

Egy m' állapotot *elérhető* az m kezdőállapotból, ha létezik $\sigma \in T^* : m[\sigma\rangle m'$ végrehajtható tüzelési sorozat. Az összes m-ből elérhető állapot halmazát $R(PN, m)$ -el jelöljük.

Korlátosság

Egy Petri-hálót *korlátos*nak nevezünk, ha minden $m \in R(PN, m_0)$ elérhető állapotban minden $p \in P$ helyre $m(p)$ korlátos.

2.1.2. Állapotegyenlet

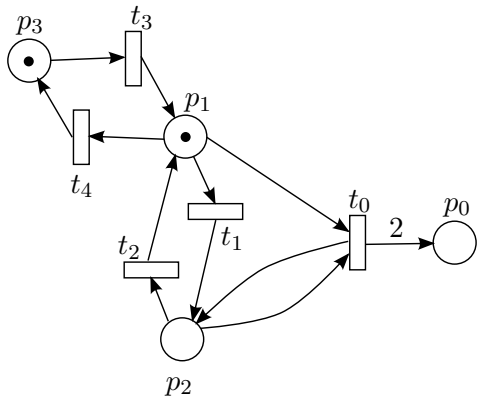
Szomszédossági mátrix

Egy Petri-háló szomszédossági mátrixa a $C_{[|P|\times|T|]}$ mátrix, ahol $|P|$ a helyek, $|T|$ a tranzíciók száma, a mátrix elemei pedig: $C(i,j) = w^+(i,j) - w^-(i,j)$. A mátrix egy $C(p_i, t_i)$ eleme tehát a $p_i \in P$ helyen lévő tokenek számának változását jelenti a $t_i \in T$ tranzíció eltüzelése esetén.

2. példa. *A 2.2. árbán látható Petri-háló szomszédossági mátrixa:*

$$\begin{bmatrix} 2 & 0 & 0 & 0 & 0 \\ -1 & -1 & 1 & 1 & -1 \\ 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & -1 & 1 \end{bmatrix}.$$

A példán az is látszik, hogy a szomszédossági mátrix nem írja le egyértelműen a Petri-hálót, ebben az esetben például a t_0 és p_2 közötti oda-vissza élek helyén a mátrix értéke 0.



2.2. ábra. Példa Petri-háló

Állapotegyenlet

Egy m állapotot értelmezhetünk egy olyan oszlopvektorként, ahol $m(p_i)$ a $p_i \in P$ helyen lévő tokenek száma. Egy $t_j \in T$ tranzíció tüzelési szám vektora egy olyan u oszlopvektor, amelynek minden eleme 0, kivéve a j. pozíciót, ahol 1 van. Ekkor ha t_j engedélyezett az m állapotban, akkor a t_j eltüzelése után kapott $m[t_j\rangle m_1$ állapot a szomszédossági mátrix definíciója miatt a következő módon is megkapható:

$$m + Cu = m_1 \tag{2.1}$$

Ezt általánosíthatjuk több tranzíció egymás utáni tüzelésére is, ahol $u_1, u_2, \ldots, u_k$ vektorok tartoznak az egyes tranzíciókhoz:

$$m + Cu_1 + Cu_2 + \ldots + Cu_k = m' \tag{2.2}$$

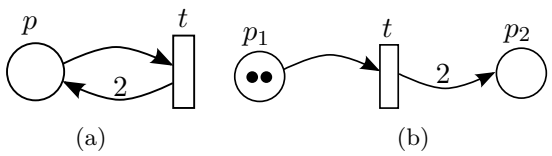
A mátrix-vektor szorzás disztributivitását felhasználva, az $u_1, u_2, \ldots, u_k$ vektorok összegét x-el jelölve kapjuk meg az állapotegyenletet, ami egy lineáris egyenletrendszer az alábbi alakban:

$$m + Cx = m' \tag{2.3}$$

Egy $x \in \mathbb{N}^{|T|}$ vektort megoldásnak nevezünk, ha kielégíti az állapotegyenletet. A megoldásvektor egy elemét $x(t)$-vel jelöljük[2], ami azt mutatja meg, hogy a t tranzíció hányszor tüzel a megoldásban. Figyeljük meg, hogy minden m-ből elérhető m' állapot esetén biztosan találunk megoldást, mert a hozzá tartozó σ végrehajtható tüzelési sorozat tüzelési szám vektora kielégíti az $m + C\wp(\sigma) = m'$ egyenletet. Fordítva ez nem feltétlenül igaz, mert egy m' állapothoz létezhet úgy x megoldásvektor, hogy x-hez nem található olyan végrehajtható tüzelési sorozat, amelynek tüzelési szám vektora megegyezne x-el. Az x megoldásvektort *realizálható*nak nevezünk, ha létezik hozzá σ végrehajtható tüzelési sorozat úgy, hogy $\wp(\sigma) = x$.

3. példa. *A 2.3(a). ábrán látható Petri háló esetén a $m = (0) \to m' = (1)$ állapotátmenetre teljesül a szükséges feltétel, ugyanis az $x = (1)$ kielégíti az állapotegyenletet. Ugyanakkor a hozzá tartozó tüzelési sorozat a t eltüzelését jelentené, ami nem engedélyezett.*

4. példa. *A 2.3(b). ábrán látható Petri háló esetén a $m = (2, 0) \to m' = (0, 4)$ állapotátmenetre teljesül a szükséges feltétel, ugyanis az $x = (2)$ kielégíti az állapotegyenletet. A t tranzíció kétszeri eltüzelése pedig végrehajtható.*



2.3. ábra. Példa hálók az állapotegyenlet kielégíthetőségére

T-invariánsok

Egy $x \in \mathbb{N}^{|T|}$ vektor *T-invariáns*, ha kielégíti a $Cx = 0$ egyenletet. A T-invariáns fontos tulajdonsága, hogy nem változtatja meg a tokeneloszlást[3] : $m + Cx = m + 0 = m$. Ha egy T-invariánshoz találunk végrehajtható tüzelési sorozatot, akkor a T-invariánst *realizálható*nak nevezzük [17]. Két realizálható T-invariáns összege is realizálható (egymás után eltüzelhetők, mert nem változtatják meg az állapotot), de ha egy realizálható T-invariáns kettő T-invariáns összege, akkor nem biztos, hogy a tagok külön-külön is realizálhatók.

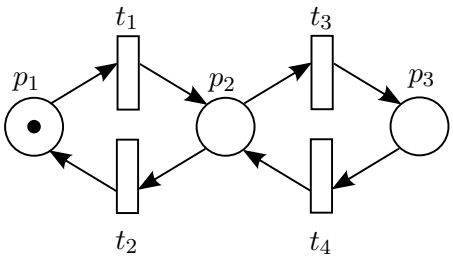
5. példa. *A 2.4. ábrán több T-invariáns is található: $T_1 = \{t_1, t_2\}, T_2 = \{t_3, t_4\}, T_3 = \{t_1, t_2, t_3, t_4\}$ és $T_1 + T_2 = T_3$. Látható, hogy T_3 realizálható t_1, t_3, t_4, t_2 sorrendben, viszont T_2 nem realizálható, mert sem t_3, sem t_4 nem engedélyezett.*

Állapotegyenlet megoldásainak tere

Egy adott PN Petri-háló $m + Cx = m'$ állapotegyenletéhez léteznek $j, k \in \mathbb{N}$ számok és véges vektorhalmazok: $B = \{b_i \in \mathbb{N}^{|T|} \mid 1 \leq i \leq j\}$ (bázisvektorok) és $P = \{p_i \in \mathbb{N}^{|T|} \mid 1 \leq i \leq k\}$ (periódusvektorok) úgy, hogy [17]:

[2]Egy $t_j \in T$ tranzíció és x megoldásvektor esetén x_j és $x(t_j)$ ugyanazt jelenti.

[3]A T-invariánsok a modellezett rendszerben gyakran egy ciklikus viselkedéshez kötődnek, ami ugyanabba az állapotba tér vissza újra és újra.



2.4. ábra. T-invariáns példa

- minden $b_i \in B$ vektor páronként nem összehasonlítható[4], azaz ilyen értelemben minimális,
- P bázist alkot a $Cx = 0$ nemnegatív megoldástere felett: $P^* = \{\sum_{i=1}^{k} n_i p_i \mid n_i \in \mathbb{N}, p_i \in P\}$,
- minden x megoldáshoz létezik $n_i \in \mathbb{N}$ ($1 \leq i \leq k$-ra) és $n \in \{1, \ldots, j\}$ úgy, hogy $x = b_n + \sum_{i=1}^{k} n_i p_i$,
- minden x megoldásra az $x + P^*$ halmaz összes vektora is megoldás.

Azaz minden megoldás felírható egy minimális bázisvektor és minimális T-invariánsok lineáris kombinációjának összegeként.

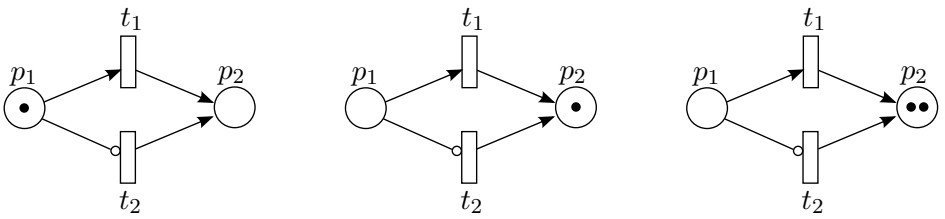
2.1.3. Kiterjesztés tiltó élekkel

A Petri-hálók kifejezőerejének növelésére számos kiterjesztést hoztak létre, többek között tiltó éles, prioritásos, önmódosító és visszaállító éles hálókat [6]. Ezek közül mi a háló tiltó élekkel történő kiegészítésével foglalkozunk. Tiltó élek segítségével ki tudjuk fejezni azt, hogy bizonyos feltételek mellett egy adott működés ne történhessen meg. Formálisan a tiltó élek halmaza egy $I \subseteq (P \times T)$ halmaz, a tiltó élekkel kiegészített Petri hálók pedig a $PN_I = (PN, I)$ struktúrával írhatók le. Tiltó éles hálók esetén a tüzelési szabály megváltozik: egy $t \in T$ tranzíció engedélyezettségéhez a korábbi feltételeken túl az is szükséges, hogy a hozzá tiltó éllel kapcsolódó helyeken ne legyen token, formálisan minden $p \in P$-re: $(p, t) \in I \Rightarrow m(p) = 0$. A tiltó élek grafikus reprezentációjában az él végpontjában nem nyíl, hanem egy üres kör van.

6. példa. *A 2.5. ábrán egy tiltó éles Petri-háló látható. A hálóban két tranzíció (t_1, t_2) és két hely (p_1, p_2) található. A 2.5(a) ábrán csak t_1 engedélyezett a (p_1, t_2) tiltó él miatt. Miután eltüzeltük t_1-et (2.5(b) ábra), csak t_2 engedélyezett. A 2.5(c) ábra a t_2 eltüzelése utáni állapotot mutatja.*

A tiltó élek bevezetése a Petri-hálók kifejezőerejét a Turing gépekével azonos szintűre emeli [6], ugyanakkor számos problémát felvet. A tiltó élek egyáltalán nem jelennek meg az állapotegyenletben, így annak megoldásakor sem vesszük őket figyelembe, ami ahhoz vezethet, hogy a kapott megoldás nem lesz realizálható. Ugyan létezik módszer arra, hogy egy tiltó éles hálót visszavezessünk egyszerű Petri-hálóra, de ez csak korlátos hálók esetén működik [15]. Számos analízis módszer is csak korlátozottan, vagy egyáltalán nem alkalmazható a tiltó éles Petri-hálókra, az elérhetőségi probléma (lásd 2.2. fejezet) pedig bizonyítottan eldönthetetlen, amennyiben a háló legalább 2 tiltó élet tartalmaz [6].

[4]Két vektor páronként nem összehasonlítható, ha mindkettőnek van olyan komponense, amelyik kisebb, mint a másik vektor azonos komponense.



(a) Kezdeti tokeneloszlás (b) Tokeneloszlás t_1 eltüzelése után (c) Tokeneloszlás t_2 eltüzelése után

2.5. ábra. Példa tiltó éles Petri-háló

2.2. Elérhetőségi probléma

A Petri-hálók elérhetőségi problémája azzal foglalkozik, hogy egy kezdőállapotból elérhető-e egy adott célállapot. Amennyiben nem egy konkrét célállapotot adunk meg, hanem csak bizonyos feltételek teljesülését írjuk elő az elérendő állapotra, akkor rész-elérhetőségről beszélünk.

2.2.1. Elérhetőségi probléma

Egy Petri-hálóból, egy kezdőállapotból és egy célállapotból álló struktúrát elérhetőségi problémának nevezünk, és formálisan $m' \in R(PN, m)$-vel jelölünk[5]. Az elérhetőségi problémára a válasz pontosan akkor „igen", ha m' szerepel az m-ből elérhető állapotok között.

2.2.2. Rész-elérhetőségi probléma

Gyakran előfordul, hogy nem egy konkrét m' állapot elérhetőségét szeretnénk vizsgálni, hanem lineáris összefüggéseket adunk meg a tokeneloszlásra vonatkozóan. Ezeket az összefüggéseket *predikátum*nak nevezzük, és a következő alakúak [9]:

$$\mathcal{P}(m) \Leftrightarrow Am \geq b, \tag{2.4}$$

ahol $\mathcal{P}(m)$ a predikátum jelölése, A együtthatómátrix, b együtthatóvektor, a predikátum pedig pontosan akkor igaz, ha a jobb oldali lineáris egyenlőtlenség-rendszer teljesül.

Az $SR(PN, m, \mathcal{P})$ rész-elérhetőségi problémára pontosan akkor „igen" a válasz, ha létezik olyan m' állapot, amire $\mathcal{P}(m')$ teljesül és m' szerepel az m-ből elérhető állapotok között.

7. példa. *Tegyük fel, hogy a 2.2. ábrán látható Petri-hálóban a p_0 helyre szeretnénk 2 tokent rakni. Ennél a hálónál a (t_1, t_3, t_0) tüzelési sorozatra teljesül, hogy utána $m(p_0) = 2$ lesz, viszont a többi hely tokeneloszlása is megváltozik. Teljes elérhetőségi probléma esetén a többi hely tokeneloszlását is meg kell adni, amit nem mindig tudunk előre kiszámolni.*

Rész-elérhetőségi problémával akár azt is meg tudjuk vizsgálni, hogy tudunk-e úgy 2 tokent rakni p_0-ra, hogy közben p_1, p_3 helyeken legalább egy token maradjon. Ez egy olyan feltétel, amelyben helyek összegére szabtunk meg egy alsó korlátot. Teljes elérhetőségi problémával ezt nem tudjuk eldönteni, pedig például a (t_1, t_3, t_0, t_2) tüzelési sorozat megfelel a feltételnek.

2.2.3. Szükséges és elégséges feltételek

Az elérhetőségi és a rész-elérhetőségi problémára is megfogalmazunk szükséges és elégséges feltételeket.

[5]A dolgozatban a példák során a kevésbé formális, de szemléletes $m \to m'$ jelölést is használjuk

Elérhetőségi probléma

Az $m' \in (PN, m)$ elérhetőségi probléma szükséges feltétele, hogy az $m + Cx = m'$ állapotegyenletnek létezzen megoldása, mert ha m-ből elérhető m', akkor létezik σ végrehajtható tüzelési sorozat amire $m[\sigma\rangle m'$ és ekkor $\wp(\sigma)$ biztosan megoldása az állapotegyenletnek. Ez a feltétel azonban nem elégséges, ahogy azt a korábban bemutatott 3. példa is tanúsítja. Elégséges feltétel például egy konkrét $\sigma : m[\sigma\rangle m'$ tüzelési sorozat megmutatása.

Rész-elérhetőségi probléma

Az $SR(PN, m, \mathcal{P})$ rész-elérhetőségi probléma szükséges feltétele, hogy létezzen olyan x vektor és m' állapot, amelyekre az $m + Cx = m'$ és $Am' \geq b$ teljesül. Az elérhetőségi problémához hasonlóan egy konkrét $\sigma : m[\sigma\rangle m'$ tüzelési sorozat itt is elégséges feltétel.

2.2.4. Komplexitás és eldönthetőség

A Petri-hálók elérhetőségi problémája a nehéz matematikai problémák közé tartozik. Az eldönthetőségét már bizonyították [12], de felső korlátot még nem sikerült adni. R. J. Lipton egy EXSPACE-nehéz alsó korlátot adott a problémára [11], azaz nem létezhet olyan eszköz, ami minden esetben hatékonyan oldja meg a problémát.

A tiltó élekkel kiegészített Petri-hálók átalakíthatók egyszerű Petri-hálókká a helyek szétbontásával, ez azonban csak korlátos hálókra alkalmazható. Általános tiltó élekkel rendelkező Petri-hálók esetén, ha legalább 2 tiltó él van a hálóban, akkor Hilbert 10. problémájára visszavezetve bizonyítható az eldönthetetlenség [6].

2.3. Lineáris Programozás (LP)

Az állapotegyenlet megoldása és a predikátumokat teljesítő állapotok keresése *lineáris programozás*i feladatra vezethető vissza. A lineáris programozás egy olyan matematikai módszer, amellyel egy matematikai modellben kereshetünk optimális megoldást bizonyos feltételek teljesülése mellett [7]. Formálisan a lineáris programozás célja, hogy egy lineáris költségfüggvényt minimalizáljuk lineáris egyenlőségek és egyenlőtlenségek teljesülése mellett. A lineáris programozási problémák kanonikus alakja:

maximalizálandó függvény: $c^T x$,
miközben $Ax \leq b$ és $x \geq 0$,

ahol x a változók vektora (amit meg kell határozni), c és b együtthatóvektorok, A pedig együtthatómátrix. A lineáris programozási probléma kielégíthető területe egy konvex poliéder, amelyet véges sok féltér metszete határoz meg. A lineáris programozás célja ezen a területen belül megkeresni azt a pontot, ahol a költségfüggvény maximális (vagy minimális), ha van ilyen.

A lineáris programozás megoldására több algoritmus is létezik, melyek komplexitása polinomiális, azaz nagyon hatékonyak.

2.3.1. Egészértékű Lineáris Programozás (ILP)

Amennyiben a keresett x változókról feltesszük, hogy egész számok legyenek, akkor a problémát *egészértékű lineáris programozás*nak nevezzük. A lineáris programozással szemben, ami legrosszabb esetben is hatékonyan megoldható, az egészértékű lineáris programozás NP-nehéz komplexitási osztályba tartozik.

Petri-hálók állapotegyenletének megoldása éppen egy ilyen egészértékű lineáris programozási probléma, ahol a keresett x vektor a tranzíciók tüzelési számait jelöli, miközben a

$Cx = m' - m$ és $x \geq 0$ feltételeknek kell teljesülnie. Ezt a feladatot a nyílt forráskódú *Lp-solve* [1] eszközre bízzuk, amelyet részletesen is bemutatunk az implementációs fejezetben (5.).

3. fejezet

CEGAR algoritmus Petri-hálókhoz

Ebben a fejezetben bemutatjuk az ellenpélda-alapú absztrakció finomítás alapelvét (lásd 3.1. fejezet) és alkalmazását Petri-hálók elérhetőségének vizsgálatára (3.2. fejezet). Megvizsgáljuk a dolgozatunk alapjául szolgáló algoritmus hiányosságait (3.3. fejezet) és bemutatjuk algoritmikus fejlesztéseinket, amelyek segítségével kiterjesztettük az algoritmussal vizsgálható problémák körét. A fejezet végén az eredeti és a saját, kibővített algoritmusunk teljességét és helyességét is megvizsgáljuk (3.5. fejezet).

3.1. CEGAR algoritmus

Egy modell állapotterén végzett elérhetőség vizsgálat során sokszor szembesülünk a lehetséges állapotok számának exponenciális növekedésével[1], vagy sok esetben az állapotok végtelen nagy számával. Az ellenpélda-alapú absztrakció finomítás (CEGAR) egy általános megközelítés, amely az állapotok számát igyekszik csökkenti, végtelen állapotterű rendszerek esetén pedig egy véges absztrakciót adni. A 3.1. ábrán látszanak a CEGAR folyamat lépései, amelyet a következőkben részletesebben bemutatunk.

Absztrakció

Az absztrakció egy olyan matematikai eszköz, melyet széles körben használnak a mérnöki problémamegoldásban. Absztrakciót a számunkra érdektelen részek elfedésére és a fontos tulajdonságok kiemelésére használunk. Így egyszerűbb, véges modellen tudjuk vizsgálni a tulajdonságok teljesülését. Többféle absztrakciós módszer is létezik, mind feltételek teljesülésének, mind pedig nem teljesülésének bizonyítására. Az általános CEGAR módszer esetén az absztrakt modell felülbecsli az eredetit (egzisztenciális absztrakció) azáltal, hogy csak a számunkra releváns információt tartja meg, a többit definiálatlanul hagyja. Az absztrakt modell többféle viselkedéssel rendelkezik, és az eredeti modell egyik viselkedése sem veszik el. Az absztrakció során az eredeti modell azon állapotait, amelyek eleget tesznek valamilyen predikátumnak, az absztrakt modellben egy összevont állapotként kezeljük. Ahogy a 3.1. ábrán is látható, a CEGAR folyamat első lépése a kezdő absztrakció és a vizsgálandó tulajdonság formulájának megadása.

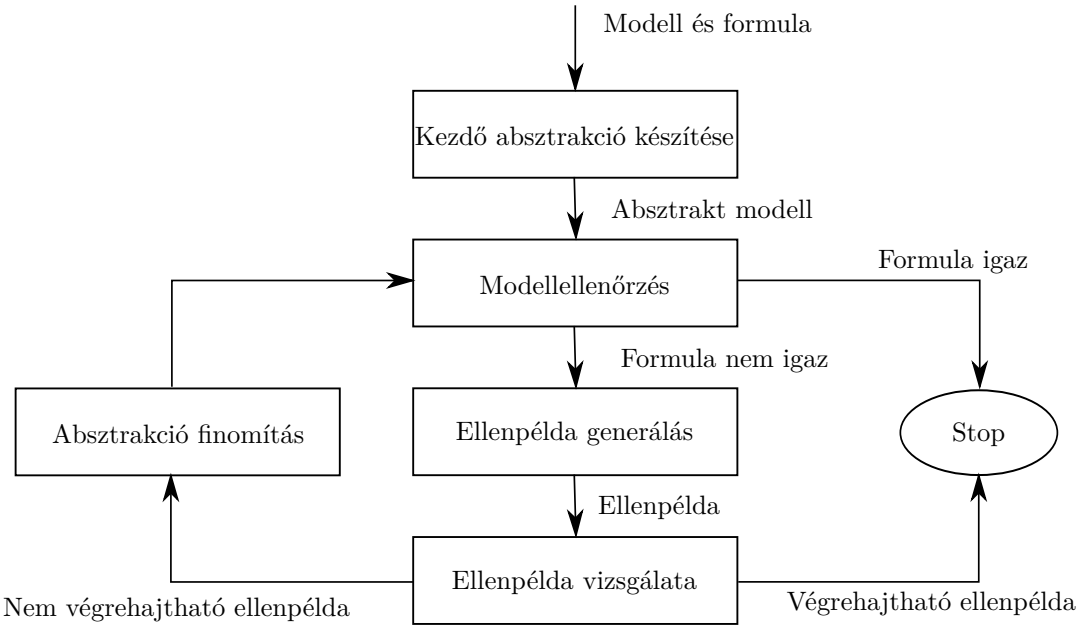
Ellenpélda-alapú finomítás

A modellellenőrzés (3.1. ábra 2. lépése) során megvizsgáljuk, hogy a formula (invariáns tulajdonság) teljesül-e az absztrakt modellünkben. Amennyiben teljesül, akkor a felülbecslés miatt az eredeti modellben is igaz a formula és kész vagyunk. A felülbecslésből adódóan

[1] Az angol szakirodalomban „state explosion”-ként említik ezt a problémát.

azonban ha valami nem igaz az absztrakt modellben, attól az eredetiben még teljesülhet. Ebben az esetben meg kell vizsgálnunk, hogy az absztrakció „durvasága” miatt nem teljesül a formula, vagy már az eredeti modellben sem volt igaz. A vizsgálathoz generálunk egy ellenpéldát, amiről eldöntjük, hogy végrehajtható ellenpélda-e.

Amennyiben az ellenpélda végrehajtható, a formula már az eredeti modellben sem igaz és kész vagyunk. Ellenkező esetben finomítani kell az absztrakción úgy, hogy kizárja ezt a nem végrehajtható ellenpéldát, és így újból ellenőrizhessük az immáron finomított absztrakt modellünket.



3.1. ábra. CEGAR folyamatábra

3.2. Petri-háló elérhetőség vizsgálata CEGAR algoritmus segítségével

Ebben az alfejezetben részletesen bemutatjuk a dolgozatunk kiindulási alapjául szolgáló algoritmust [17], amely a Petri-hálók elérhetőségi problémájának eldöntésére a CEGAR megközelítést alkalmazza.

3.2.1. CEGAR megközelítés Petri-hálókra

Petri-hálók elérhetőségi problémájának CEGAR megközelítéssel történő vizsgálatakor a kiinduló absztrakciónak az állapotegyenletet tekintjük. Az állapotegyenlet egy lineáris egyenletrendszer, aminek kielégíthetősége szükséges, de nem elégséges feltétele az elérhetőségnek, és emiatt megoldása felülbecsli a ténylegesen elérhető állapotokat. A megoldáshoz használt ILP eszköz egy adott célfüggvény szerinti minimális megoldást ad. A mi esetünkben a célfüggvényben minden változó (tranzíció) együtthatója 1, hogy az ILP megoldó a lehető legrövidebb tüzelési sorozatokat adja vissza. A megoldás során a következő esetek merülhetnek fel:

- Ha az állapotegyenlet kielégíthetetlen, akkor a szükséges feltétel miatt a célállapot nem elérhető.

- Ha találunk megoldást, akkor meg kell vizsgálni, hogy létezik-e olyan tüzelési sorozat, amely eltüzelhető és Parikh képe megegyezik a megoldással.
  - Ha találunk ilyen tüzelési sorozatot, akkor ez már egy elégséges feltétel, tehát a célállapot elérhető.
  - Ha nem létezik megfelelő tüzelési sorozat, akkor az előbbi megoldásvektor egy ellenpélda, amit az absztrakció finomításával ki kell zárni, hogy az ILP eszköz más megoldást tudjon produkálni.

Az absztrakció finomítása során a cél az, hogy az ellenpéldát úgy zárjuk ki a lehetséges megoldások közül, hogy közben realizálható megoldás ne vesszen el. Ehhez a tranzíciók tüzelési számára vonatkozó lineáris egyenlőtlenségeket, ún. *megkötés*eket használunk, melyeket a következő alfejezetben (3.2.2.) ismertetünk. Az egyenlőtlenségek hozzáadása után újból meg tudjuk oldani az immár kiegészített állapotegyenletet.

3.2.2. Megkötések és részleges megoldások

Megkötések

Egy $PN = (P, T, E, W)$ Petri-háló absztrakciójának finomításához kétféle megkötést definiálunk. Ezek a tranzíciók tüzelési számára vonatkozó lineáris egyenlőtlenségek a következő alakban:

- az *ugró megkötés*ek $|t_i| < n$ alakúak, ahol $n \in \mathbb{N}$, $t_i \in T$ és $|t_i|$ a t_i tranzíció tüzelési száma. Az ugró megkötés jelentése az, hogy olyan megoldásokat keresünk, amelyben t_i tranzíció kevesebbszer tüzel, mint n. Segítségükkel a megoldások terében a bázisvektorok között tudunk ugrálni, kihasználva azt, hogy azok páronként nem összehasonlíthatók.

- a *növelő megkötés*ek $\sum_{i=1}^{k} n_i|t_i| \geq n$ alakúak, ahol $n_i \in \mathbb{Z}$, $n \in \mathbb{N}$ és $t_i \in T$. A növelő megkötés jelentése az, hogy olyan megoldásokat keresünk, amelyben a tranzíciók tüzelési számainak súlyozott összege legalább n. Ezáltal nem minimális megoldásokat is megkaphatunk.

Az ilyen jellegű megkötéseket arra használjuk, hogy kizárjunk bizonyos megoldásokat, ezáltal elérhetjük, hogy az ILP eszköz másik minimális, vagy akár nem minimális megoldást produkálhasson.

Mielőtt rátérnénk a megkötések készítésének pontos módjára, bevezetjük a *részleges megoldás*ok fogalmát. Részleges megoldásokat az állapotegyenletből és megkötésekből álló lineáris egyenlőtlenség-rendszer egy megoldásához tudunk generálni úgy, hogy annyi tranzíciót tüzelünk el, amennyit csak lehetséges. A tüzelések számának egyrészt az szab korlátot, hogy egy tranzíciót nem akarunk többször eltüzelni, mint amennyiszer a megoldásban szerepel, másrészt előfordulhat, hogy néhány tranzíció eltüzelése után nem marad engedélyezett tranzíció.

Részleges megoldások

Legyen $PN = (P, T, E, W)$ Petri-háló, $m' \in R(PN, m)$ elérhetőségi probléma és Ω egy olyan teljes rendezés $\mathbb{N}^{|T|}$ felett, ahol $x < y$ akkor, ha $\sum_{t \in T} x(t) < \sum_{t \in T} y(t)$ teljesül. Az elérhetőségi probléma egy részleges megoldása egy olyan $(\mathcal{C}, x, \sigma, r)$ struktúra, ahol:

- $\mathcal{C}$ az ugró és növelő megkötéseket tartalmazó halmaz. Az állapotegyenlet és $\mathcal{C}$ megkötései alkotják a megoldandó ILP problémát.

- x az Ω rendezés szerinti minimális megoldás, amely kielégíti az előbbi ILP problémát, azaz az állapotegyenletet és $\mathcal{C}$ összes megkötését.
- $\sigma \in T^*$ maximális végrehajtható tüzelési sorozat, amire $\wp(\sigma) \leq x$ teljesül, azaz minden tranzíció legfeljebb annyiszor tüzelhet, mint ahányszor a megoldásvektorban szerepel. A σ tüzelési sorozat olyan értelemben maximális, hogy amíg van olyan tranzíció ami engedélyezett és kevesebbszer tüzeltük el, mint ahányszor megoldásvektorban szerepel, akkor el kell tüzelni.
- $r = x - \wp(\sigma)$ maradékvektor, melynek i. eleme megmutatja, hogy az i. tranzíciót hányszor kellene még eltüzelni. A tüzelési sorozatok maximalitása miatt a maradékvektorban nem 0 együtthatóval szereplő tranzíciók nem lehetnek engedélyezettek (hiszen akkor eltüzelhetnénk őket).

Egy részleges megoldást *teljes megoldás*nak nevezünk, ha az $r = 0$ feltétel teljesül. Ekkor $\wp(\sigma) = x$, azaz σ egy olyan tüzelési sorozat, amire $m[\sigma\rangle m'$. A teljes megoldás tehát elégséges feltétele az elérhetőségi problémának. Figyeljük meg, hogy tetszőleges végrehajtható tüzelési sorozathoz, amely a kiinduló állapotból a célállapotba visz, található teljes megoldás:

1. Következmény. *(A realizálható megoldásokhoz tartozik teljes megoldás).* Az állapotegyenlet minden (σ tüzelési sorozat által) realizálható x megoldásához található egy olyan $(\mathcal{C}, x, \sigma, 0)$ teljes megoldás, ahol $\wp(\sigma) = x$ és $\mathcal{C}$ minden olyan t_i tranzícióhoz, amire $x(t_i) > 0$ teljesül, tartalmaz egy olyan növelő megkötést, ahol t_i együtthatója 1, a többi tranzícióé 0 és $n = x(t_i)$, azaz $|t_i| \geq x(t_i)$ alakú.

Bizonyítás. Mivel x realizálható σ által, ezért biztosan megoldása az állapotegyenletnek. Az $|t_i| \geq x(t_i)$ alakú növelő megkötések miatt x-nél Ω-kisebb megoldása nincs az állapotegyenletnek, tehát x az Ω-minimális megoldás, amelyhez a σ tüzelési sorozattal találunk egy $r = 0$ maradékvektorú teljes megoldást. □

Az előbbi állítás kimondja, hogy minden realizálható x megoldásra elérhetjük megkötésekkel, hogy az ILP eszköz azt adja ki. Általában többféle megkötés-halmaz is képes ugyanazt az x megoldásvektort adni, akár ugró és növelő megkötéseket vegyesen is. A lényeg, hogy mindig létezik legalább egy ilyen megkötés-halmaz. A kérdés az, hogy a teljes megoldást el tudjuk-e érni úgy, hogy a kezdetben megkötések nélküli állapotegyenlet minimális megoldását bővítjük lépésenként megkötésekkel.

1. Lemma. *(Út egy teljes megoldásig).* Legyen b az $m' \in R(PN, m)$ elérhetőségi probléma állapotegyenletének Ω-minimális megoldása és $ps' = (\mathcal{C}_l, b' + \sum_{i=1}^{k} n_i p_i, \sigma, 0)$ egy teljes megoldás. Legyen továbbá $\mathcal{C}_l$ megkötéslista l elemű, ahol az egyes megkötéseket c_j-vel $(1 \leq j \leq l)$ jelöljük. A megoldásvektor egy bázisvektor (b') és T-invariánsok (p_i) lineáris kombinációjának összege, σ egy végrehajtható tüzelési sorozat, a maradékvektor pedig nullvektor. Ekkor minden n-re $(0 \leq n \leq l)$ létezik $ps_n = (\mathcal{C}_n, x_n, \sigma_n, r_n)$ részleges megoldás úgy, hogy $\mathcal{C}_n = \{(c_j) | 1 \leq j \leq n\}$ és $ps_0 = (\emptyset, b, \sigma_0, r_0)$, $ps_l = ps'$ és $x_{n_1} \leq_\Omega x_{n_2}$, ha $n_1 \leq n_2$.

Kevésbé formálisan ez azt jelenti, hogy egy l darab megkötést tartalmazó megkötéslistájú teljes megoldáshoz el tudunk jutni l db részleges megoldáson keresztül úgy, hogy a kezdeti megkötések nélküli részleges megoldás megkötéseit bővítjük, amíg el nem érünk a teljes megoldáshoz. Azt, hogy pontosan milyen megkötéseket és hogyan adunk hozzá, majd később tárgyaljuk, egyelőre belátjuk, hogy a fenti lemma igaz.

Bizonyítás. Legyen ps_{n_1} és ps_{n_2} két köztes részleges megoldás $(0 \leq n_1 \leq n_2 \leq l)$. Tudjuk, hogy x_{n_2} kielégíti az állapotegyenletet és $\mathcal{C}_{n_2}$ t. Mivel $\mathcal{C}_{n_1} \subseteq \mathcal{C}_{n_2}$ teljesül[2], ezért

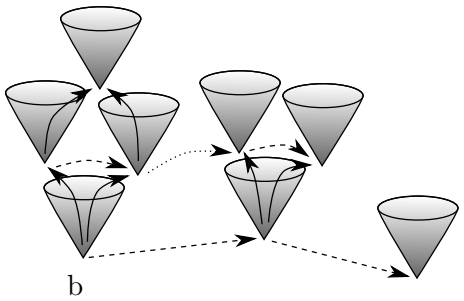
[2]Részhalmaz alatt itt azt értjük, hogy $\mathcal{C}_{n_1}$ megkötései egy részhalmazát alkotják $\mathcal{C}_{n_2}$ megkötéseinek.

x_{n_2} megoldása az állapotegyenletnek és $\mathcal{C}_{n_1}$ megkötés-halmaznak is. Az állapotegyenletből és $\mathcal{C}_{n_1}$ megkötés-halmazból álló egyenlőtlenség-rendszer Ω-minimális megoldása x_{n_1}, ezért $x_{n_1} \leq_\Omega x_{n_2}$ teljesül. Ezt a gondolatot általánosíthatva minden n_i-re kapjuk, hogy: $b \leq_\Omega x_1 \leq_\Omega \ldots \leq_\Omega x_l$. Tudjuk, hogy $x_l = b' + \sum_{i=1}^{k} n_i p_i$ egy létező megoldása a legszigorúbb egyenlőtlenség-rendszernek (azaz az állapotegyenletnek és $\mathcal{C}_l$ megkötés-halmaznak), ebből következően minden olyan egyenlőtlenség-rendszer megoldható, ami az állapotegyenletből és $\mathcal{C}_l$ valamely részhalmazából áll. Ekkor minden köztes megoldásvektor (x_{n_i}) létezik, így találunk köztes ps_{n_i} részleges megoldásokat úgy, hogy x_{n_i}-ből annyi tranzíciót tüzelünk el, amennyit csak tudunk. □

Tekintsünk most egy $ps = (\mathcal{C}, x, \sigma, r)$ részleges megoldást, ami nem teljes, azaz $r \neq 0$. Ez nyilvánvalóan azt jelenti, hogy néhány tranzíció nem tud elégszer eltüzelni. Ekkor a következő három lehetőségünk van:

1. előfordulhat, hogy x realizálható egy másik σ' tüzelési sorozattal, amelyre $\wp(\sigma') = x$, azaz létezik egy $ps' = (\mathcal{C}, x, \sigma', r)$ teljes megoldás.
2. Egy ugró megkötést hozzáadva egy Ω-nagyobb megoldást kaphatunk, amelyből újabb részleges megoldásokat generálhatunk.
3. Ha van olyan $t \in T$ tranzíció, amire $r(t) > 0$, akkor növelő megkötés segítségével megnövelhetjük t bemeneti helyein a tokenek maximális számát. Mivel a végső m' állapot ugyanaz marad, ez tokenek kölcsönvételét jelenti bizonyos helyekre. Ezt olyan T-invariánsokkal érhetjük el, amelyek átmennek az adott hely(ek)en.

A fenti ötleteket a 3.2 ábra vizuálisan mutatja be. A „kúpok" megoldásokat jelölnek a fölöttük lévő lineáris térrel, amelyeknek akár közös részük is lehet. Az Ω-minimális megoldást b jelöli. A folytonos nyilak növelő megkötések, a szaggatott nyilak ugró megkötések hozzáadását jelölik. Ilyen ugrások felsőbb szinteken is előfordulhatnak, ahogy azt a pontozott nyíl mutatja. A következő fejezetben bemutatjuk, hogy hogyan lehet úgy megkötéseket készíteni, hogy az általunk kívánt hatást érjék el az állapotegyenlet megoldásainak bejárása során.



3.2. ábra. Állapotegyenlet megoldásainak tere

3.2.3. Megkötések generálása

A részleges megoldások ismertetése után bemutatjuk, hogy hogyan kell ugró és növelő megkötéseket generálni úgy, hogy elérjünk a teljes megoldáshoz (amennyiben van).

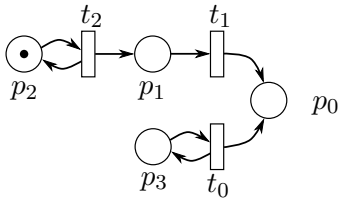
Ugró megkötések

Először bebizonyítjuk, hogy ugró megkötésekkel bármelyik minimális bázisvektort megkaphatjuk.

2. Lemma. *(Ugrás bázisvektorok között).* Legyenek $b, b' \in B$ bázisvektorai az állapotegyenlet és a $\mathcal{C}$ megkötés-halmaz megoldásterének. Legyen továbbá b az Ω-minimális megoldás. Ekkor elérhetjük, hogy az ILP eszköz b'-t adja megoldásként úgy, hogy $|t_i| < n_i$ $(n_i \in \mathbb{N})$ alakú ugró megkötéseket adunk hozzá egymás után a $\mathcal{C}$ megkötés-halmazhoz.

Bizonyítás. Tudjuk, hogy $b \leq_\Omega b'$ teljesül, de mivel b' is minimális megoldás, $b \leq b'$ nem teljesülhet, mert kell lennie egy komponensnek, amiben b' a kisebb. Ekkor létezik egy olyan $t_i \in T$ tranzíció, amelyre $b'(t_i) < b(t_i)$. Ha hozzáadjuk a $|t_i| < b(t_i)$ ugró megkötést $\mathcal{C}$-hez, akkor b-t már nem kaphatjuk meg megoldásként, mert nem elégíti ki az előbbi megkötést. Legyen a kapott új megoldás b''. Ha $b'' = b'$ akkor kész vagyunk, egyébként b'-t még nem zártuk ki a megoldások közül, mivel $|t_i| < b(t_i)$ teljesül, azaz még kielégíti az állapotegyenletet és a megkötéseket. Ekkor $b'' \leq_\Omega b'$ teljesül, és rekurzívan alkalmazhatjuk az előbbi gondolatmenetet $b := b''$ helyettesítéssel. Mivel csak véges számú megoldás lehet Ω-kisebb b'-nél, ezért az algoritmus véges számú lépésben eljut b'-ig. □

8. példa. *Tekintsük a 3.3. ábrán látható Petri-hálót a* $(0,0,1,0) \to (1,0,1,0)$ *elérhetőségi problémával. Ekkor az* Ω*-minimális megoldásvektor* $x = (1,0,0)$*, azaz* t_0 *eltüzelése. Ez nem realizálható, azonban a* $|t_0| < 1$ *ugró megkötéssel megkaphatunk egy* $x' = (0,1,1)$ *másik minimális megoldásvektort, amely* t_2, t_3 *sorrendben realizálható.*



3.3. ábra. Ugró megkötés példa

Figyeljük meg, hogy ezzel az „ugrálós" módszerrel nem minimális megoldások nem érhetőek el, hiszen a „létezik egy olyan $t_i \in T$ tranzíció, amelyre $b'(t_i) < b(t_i)$" állítás nem mindig igaz. Erre a célra növelő megkötéseket kell használni, azonban előfordulhat, hogy az ugró és növelő megkötések ellentmondanak egymásnak. Az ugró megkötéseket csak arra használjuk, hogy egy másik minimális vektort megkapjunk, de a továbbiakban nem szeretnénk ha a tranzíciók tüzelési számait felülről korlátoznák. Tegyük fel, hogy az állapotegynletnek van egy $b' + p$ alakú megoldása, ahol $p \in P$ periódusvektor és $b' \in B$ eléréséhez legalább egy $|t_i| < n_i$ alakú ugró megkötést kell használni. Amennyiben p-ben elég sokszor szerepel t_i, azt fogjuk tapasztalni, hogy $(b' + p)(t_i) \geq n_i$ áll fenn, azaz többször kell eltüzelni, mint ahányszor az ugró megkötés engedné. Emiatt $b' + p$ nem elégíti ki az állapotegyenletből és a $|t_i| < n_i$ megkötésből álló lineáris egyenlőtlenség-rendszert, azaz egy olyan növelő megkötés, amelyben t_i elég sokszor szerepel, kielégíthetetlenné teszi a lineáris egyenlőtlenség-rendszerünket. Ennek elkerülésére az ugró megkötéseket át kell alakítani növelő megkötésekké.

3. Lemma. *(Ugró megkötések átalakítása).* Legyen z az Ω-minimális megoldása az $m + Cx = m'$ állapotegyenletből és $\mathcal{C}$ megkötés-halmazból álló lineáris egyenlőtlenségrendszernek. Álljon $\mathcal{C}'$ az összes $\mathcal{C}$-beli növelő megkötésből és $|t_i| \geq z(t_i)$ alakú növelő megkötésből minden $t_i \in T$ tranzícióra. Ekkor minden $y \geq z$-re y pontosan akkor megoldása az állapotegyenletnek és $\mathcal{C} \cap \mathcal{C}'$ megkötés-halmaznak, ha megoldása az állapotegyenletnek és $\mathcal{C}'$-nek. Továbbá nem létezik z-nél Ω-kisebb megoldása az állapotegyenletnek és $\mathcal{C}'$-nek.

Bizonyítás. Legyen $y \geq z$ megoldása az állapotegyenletnek és $\mathcal{C} \cap \mathcal{C}'$ megkötés-halmaznak. A $\mathcal{C}'$-beli további megkötések $y(t) \geq z(t)$-t írják elő, ami nyilvánvalóan teljesül. A másik irány triviális, hiszen ha y egy szigorúbb megkötésthalmazt ($\mathcal{C}'$) kielégít, akkor a kevésbé szigorút ($\mathcal{C}$) is. A második állítás belátásához legyen $z' \leq_\Omega z$, $z' \neq z$ megoldása az állapotegyenletnek és $\mathcal{C}$ megkötés-halmaznak. Ω miatt $\sum_t z'(t) \leq \sum_t z(t)$ teljesül, de mivel $z \neq z'$ ezért van legalább egy $t_i \in T$ tranzíció, amire $z'(t_i) < z(t_i)$. Ekkor a $\mathcal{C}'$-beli $|t_i| \geq z(t_i)$ feltétel miatt z' nem lehet megoldása az állapotegyenletnek és $\mathcal{C}'$-nek. □

A fenti lemma következményeként, ha egy z megoldás feletti $z + P^*$ „kúp" megoldásai érdekelnek minket, akkor az eddigi ugró megkötéseket a fent említett módon növelő megkötésekre cserélhetjük. Az ILP eszköz z-t fogja Ω-minimális megoldásként adni $\mathcal{C}'$ megkötés-halmazra is, viszont $\mathcal{C}'$-höz már nyugodtan adhatunk növelő megkötéseket.

Növelő megkötések

Legyen $ps = (\mathcal{C}, x, \sigma, r)$ egy olyan részleges megoldás, ahol $r > 0$, azaz nem teljes megoldás. Szeretnénk meghatározni azokat a helyeket, ahova még tokent kellene termelni (és a szükséges tokenek számát) ahhoz, hogy az r maradékvektorban lévő tranzíciókat el tudjuk tüzelni. Ez a probléma nehezebb, mint eldönteni, hogy egy megoldás realizálható-e, azaz, hogy nulla extra token elég-e. Alkalmazhatnánk rekurziót, de az nem lenne túl hatékony, mert egy x megoldásvektorhoz sokféle r maradékvektor tartozhat. Ugyan a maradékvektor kisebb, mint a megoldásvektor, de a rekurziós lépések száma exponenciálisan nőhet x méretéhez ($\sum_t x(t)$) képest.

Az algoritmus szerzői [17] a probléma megoldására más módszert használnak. Keresünk egy jó heurisztikát, amellyel meg tudjuk becsülni a tokenek szükséges számát. Ha helyek egy halmazára n ($n > 0$) token kell, akkor az algoritmus 1 és n között fog egy becslést adni. Amennyiben alulbecsüljük a szükséges tokenek számát, akkor kapunk egy új részleges megoldást, amire újból alkalmazhatjuk a becslésünket, így lassan konvergálhatunk a tényleges tokenszám felé. Erre a célra egy 3 lépésből álló algoritmust mutatunk be:

1. Az algoritmus első lépéseként egy függőségi gráf segítségével meghatározza azokat a helyeket és tranzíciókat amelyek számunkra érdekesek. Ezek olyan tranzíciók amelyeket szeretnénk eltüzelni, de nem tudunk és azok a helyek, amelyek miatt nem tudnak tüzelni.

2. A következő lépésben megbecsüljük, hogy minimálisan hány tokent kell ezekre a helyekre termelni, hogy esélyünk legyen eltüzelni a kívánt tranzíciókat.

3. Az utolsó lépésben az adott helyekre termelendő tokenszám ismeretében meghatározzuk a növelő megkötést.

Az első lépésben tehát meg kell határoznunk azokat a helyeket és tranzíciókat, amelyek a további vizsgálatok során érdekesek számunkra. Ehhez egy - a részleges rendezésből is ismert [16] - függőségi gráfot építünk fel az 1. algoritmus alapján.

A gráfban azok a tranzíciók szerepelnek, amelyek nem tudnak tüzelni, illetve minden ilyen tranzícióhoz tartozik legalább egy hely, ami miatt nem tud tüzelni. A gráf élei az irányuktól függően mást jelentenek:

- Egy p helyből t tranzícióba mutató él azt jelenti, hogy t tranzíció p hely miatt nincs engedélyezve.
- Egy t tranzícióból p helybe mutató él azt jelenti, hogy t tüzelése megnövelné a p helyen lévő tokenek számát.

Algoritmus 1: Függőségi gráf

bemenet: Elérhetőségi probléma $m' \in R(PN, m)$; részleges megoldás $ps = (\mathcal{C}, x, \sigma, r)$

kimenet : (P_i, T_i, X_i) struktúrák halmaza, ahol $P_i \subseteq P$, $T_i \cup X_i \subseteq T$

1 $\hat{m}$ kiszámolása: $m[\sigma\rangle\hat{m}$;
2 $G = (P_0 \cup T_0, E)$ páros gráf építése;
3 $T_0 := \{t \in T \mid r(t) > 0\}$;
4 $P_0 := \{p \in P \mid \exists t \in T_0 : W(p,t) > \hat{m}(p)\}$;
5 $E = \{(p,t) \in P_0 \times T_0 \mid W(p,t) > \hat{m}(p)\} \cup \{(t,p) \in T_0 \times P_0 \mid W(t,p) > W(p,t)\}$;
6 G erősen összefüggő komponenseinek (SCC) megkeresése;
7 i:=1;
8 **foreach** *forrás SCC (aminek nincs bemenő éle)* **do**
9 $\quad P_i := SCC \cap P_0$;
10 $\quad T_i := SCC \cap T_0$;
11 $\quad X_i := \{t \in T_0 \backslash SCC \mid \exists p \in P_i : (p,t) \in E\}$;
12 $\quad$ i:=i+1;
13 **end**

Számunkra a gráf erősen összefüggő komponensei (SCC) érdekesek, ugyanis ezek olyan helyekből és tranzíciókból álló halmazokat jelölnek, ahol a tranzíciók kölcsönösen képesek lehetnek engedélyezni egymást, amennyiben valamely helyre, vagy helyekre elég token kerülne. Egy bemenő él nélküli, azaz forrás SCC ebből következően más SCC-k tranzíciói által nem kaphat tokent. Ez azt jelenti, hogy a tokeneket máshonnan kell beszerezni, ami olyan tranzíciók tüzelését követeli meg amelyek nincsenek benne az r maradékvektorban. Minden forrás SCC helyein (P_i) és tranzícióin (T_i) kívül egy harmadik X_i halmazba felvesszük azokat a tranzíciókat, amelyek engedélyezettsége az adott SCC-től függ. Kimondhatjuk tehát a következő lemmát:

4. Lemma. Az 1. algoritmus meghatározza azokat az erősen összefüggő komponenseket (P_i,T_i), amelyekre tokent kell termelni ahhoz, hogy a részleges megoldás r maradékvektora engedélyezetté válhasson. Meghatározza továbbá azon tranzíciók halmazát (X_i), amelyek ettől az SCC-től függenek.

Egy adott (P_i, T_i, X_i) struktúrához nehéz megbecsülni pontosan a szükséges tokenszámot, ugyanis egy T_i beli tranzíció engedélyezése később akár minden $T_i \cup X_i$-beli tranzíciót engedélyezhet. A 2. algoritmus egy jó heurisztikát valósít meg a szükséges tokenszám becslésére.

5. Lemma. A 2. algoritmus helyes abban az értelemben, hogy nem becsüli felül a szükséges tokenek számát, tehát nem zár ki teljes megoldásokat a keresésből. A fenti algoritmus az első lépésben meghatározott minden P_i halmazhoz megbecsli a szükséges tokenszámot. Tehát a becslés értéke bármennyi lehet egy és a ténylegesen szükséges tokenek száma között.

Bizonyítás. A tokenszám megbecslésekor két fő esetet különböztetünk meg:

- Ha a T_i halmaz nem üres, akkor a kölcsönös függések miatt előfordulhat, hogy egy t tranzíció engedélyezetté tétele után az összes tranzíció el tud tüzelni, mivel t eltüzelése növeli valamely P_i-beli helyek tokenszámát. Az algoritmus ilyenkor megkeresi azt a tranzíciót, amelynek az engedélyezéséhez a legkevesebb token hiányzik.

Algoritmus 2: Szükséges tokenek számának becslése

bemenet: (P_i, T_i, X_i) struktúra; $m' \in R(PN, m)$ elérhetőségi probléma; $\hat{m}$ az előző lépésből

kimenet : Szükséges n tokenszám a P_i halmaz helyeire

```
1  if T_i ≠ ∅ then  n := min_{t∈T_i}(Σ_{p∈P_i} max{0, (W(p,t) − m̂(p))});
2  else
3  |  G_j csoportok létrehozása: G_j := {t ∈ X_i | W(t,p) = j}, ahol P_i = {p};
4  |  n := 0;
5  |  c := 0;
6  |  for j csökkenő ciklus 0-ig do
7  |  |  if G_j ≠ ∅ then
8  |  |  |  c := c + j + Σ_{t∈G_j}(W(p,t) − j);
9  |  |  |  if c > 0 then n := n + c;
10 |  |  |  c := −j;
11 |  |  end
12 |  end
13 |  n := n − m̂(p);
14 end
```

- Ha a T_i halmaz üres, akkor a gráf párossága miatt P_i egyetlen p_i helyből áll. Ez azt jelenti, hogy az összes X_i-beli tranzíció szükséges tokenszámát a p_i helynek ki kell elégítenie. Az X_i tranzíciói összességében csökkentik p_i tokenszámát, azonban a becsléskor figyelembe kell venni, hogy pontosan mennyit vesznek el, és raknak vissza. A tranzíciókat a visszarakott tokenszám (j) szerint csoportokba rendezzük. Amennyiben a legkisebb j értékű tranzíciókat tüzeljük el utoljára, a p_i-n megmaradó tokeneket minimalizáljuk. Az egyforma j értékkel rendelkező tranzíciókat egyszerre feldolgozhatjuk úgy, hogy mindegyik $(p_i, t) - j$ tokent vesz el, kivéve az elsőt, amelyhez szükség van plusz j tokenre. (Az első előtt nincs senki, aki visszarakna neki j db tokent.) A G_j csoport tranzíciói által meghagyott tokeneket a következő csoport felhasználhatja, ezt a c változóban jegyezzük meg. Összességében tehát megkapjuk azt a minimális tokenszámot, ami az összes X_i-beli tranzíció eltüzelését lehetővé teszi.

Figyeljük meg, hogy az algoritmus mindig egy pozitív számot ad eredményül, ugyanis mindig kell lennie legalább egy tranzíciónak a $T_i \cup X_i$ halmazban, különben nem lenne olyan tranzíció, amely P_i helyei miatt nem tud tüzelni, és ilyenkor P_i-t ki se számoltuk volna. Ha T_i nem üres, akkor az algoritmus a tranzíciókhoz hiányzó tokenek számát minimalizálja, ami egy pozitív egész. Ha T_i üres, akkor a ciklus első lépésében c pozitív lesz és végül az eredmény (n) is. □

A helyek (P_i) és a szükséges tokenszám (n) ismeretében már elő tudjuk állítani a növelő megkötést. Az állapotegyenlet változói tranzíciók, ezért a helyekre alkotott feltételünket az alábbi lemma segítségével át kell alakítani úgy, hogy tranzíciókra vonatkozzon:

6. Lemma. Legyen $PN(P, T, E, W)$ Petri-háló, $m' \in R(PN, m)$ elérhetőségi probléma, $ps = (\mathcal{C}, x, \sigma, r)$ részleges megoldás $r > 0$ maradékvektorral és $\hat{m}$ a σ eltüzelésével kapott állapot $(m[\sigma\rangle\hat{m})$. Legyen P_i a helyek halmaza, ahova n tokent kell termelni, továbbá legyen $T_i := \{t \in T \mid r(t) = 0 \wedge \sum_{p \in P_i}(W(t,p) - W(p,t)) > 0\}$, azaz olyan tranzíciók halmaza, amelyek nincsenek benne a maradékvektorban és tokent termelnek P_i helyeire. Ekkor a c

növelő megkötés a következő alakú:

$$\sum_{t\in T_i}\sum_{p\in P_i}(W(t,p)-W(p,t))|t| \geq n+\sum_{t\in T_i}\sum_{p\in P_i}(W(t,p)-W(p,t))\wp(\sigma)(t) \tag{3.1}$$

Ekkor ha az állapotegyenletből és $\mathcal{C}\cup\{c\}$ -ből álló lineáris egyenletrendszerünkhöz az ILP eszköz egy $x+y$ megoldást tud generálni (y egy T-invariáns), akkor találunk egy $ps' = (\mathcal{C}\cup c, x+y, \sigma\tau, r+z)$ részleges megoldást, ahol $\wp(\tau)+z = y$, továbbá $\sum_{t\in T_i}\sum_{p\in P_i}(W(t,p)-W(p,t))y(t) \geq n$.

Kevésbé formálisan ez azt jelenti, hogy a megkötés miatt egy y T-invariánst vesz hozzá az algoritmus a megoldáshoz, melyből valamennyi tranzíciót el is tud tüzelni, ezt jelöli a τ tüzelési sorozat. Az invariánsból el nem tüzelt tranzíciók (z) a maradékvektorhoz adódnak. A kapott plusz tokenek pedig az $x+y$ megoldásvektorban csak az y-ból jöhetnek, ezt fejezi ki az utóbbi egyenlőtlenség.

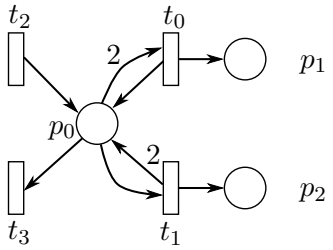
Bizonyítás. Figyeljük meg, hogy a T_i-beli tranzíciók több tokent tesznek P_i helyeire, mint amennyit elvesznek. T_i-ben csak olyan tranzíciók szerepelnek, amelyekre $r(t) = 0$, ugyanis nem szeretnénk hozzávenni olyan tranzíciókat, amelyek eddig sem tudtak eltüzelni. A megkötés bal oldalán minden $t \in T_i$ tranzíció szerepel olyan együtthatóval, mint amennyi plusz tokent termel P_i helyeire. Amennyiben a bal oldalra egy konkrét u Parikh képet helyettesítenénk (t helyére $u(t)$-t írva), akkor megkapnánk a T_i tranzíciók által P_i helyeire termelt tokenek számát u tüzelésekor. Természetesen a T_i-n kívüli tranzíciók ezt a számot csökkenthetik. A megkötés jobb oldalán kiszámoljuk, hogy T_i tranzíciói ténylegesen hány tokent termeltek eddig a σ tüzelési sorozat által, és ehhez adjuk hozzá a szükséges n tokenszámot, amit el szeretnénk érni. Mivel az $x + y$ megoldásban az extra tokenek x-ből nem jöhetnek, ezért y-nak kell őket megtermelni, azaz $\sum_{t\in T_i}\sum_{p\in P_i}(W(t,p)-W(p,t))y(t) \geq n$. σ tüzelése után előfordulhat, hogy y egy részét el tudjuk tüzelni, ebből adódik, hogy: $\wp(\tau) + z = y$. □

Amikor ezt az új c megkötést hozzávesszük az eddigiekhez, akkor a hálóban lévő T-invariánsoktól függ, hogy a szükséges n tokenből mennyit tudunk megkapni. Az állapotegyenlet és az új megkötés-halmaz megoldása után a következő 3 eset fordulhat elő:

- Ha a lineáris egyenlőtlenség-rendszer kielégíthetetlenné válik, akkor ezt a részleges megoldást nem lehet teljes megoldássá bővíteni, és nem kell továbbá foglalkozni vele.
- Ha az új részleges megoldással nem jutottunk közelebb a teljes megoldáshoz, akkor ha ezt nem vesszük észre, az algoritmus végtelen ciklusba kerülhet, de megoldás nem veszik el.
- Ha a kapott részleges megoldásban sikerült néhány tranzíciót eltüzelni a maradékvektorból, akkor további megkötéseket alkalmazhatunk. Amennyiben létezik teljes megoldás, akkor azt meg tudjuk találni ugró és növelő megkötések segítségével, mert csak olyan megkötéseket adunk hozzá, amelyek mindenképp szükségesek. Az ugró megkötésekre azért van szükség, mert több olyan minimális megoldás is lehet, amely kielégíti a legutóbb hozzáadott megkötést.

9. példa. *Tekintsük a 3.4. ábrán látható Petri-hálót a $(0,0,0) \to (0,1,1)$ elérhetőségi problémával. Minimális megoldásvektor az $x = (1,1,0,0)$, azaz t_0 és t_1 eltüzelése. Mivel egyik tranzíció sem engedélyezett, kereshetünk növelő megkötést. A függőségi gráfba fel kell venni t_0-t, t_1-et és p_0-t, mert miatta nem tudnak tüzelni. p_0-ból t_0-ba és t_1-be is mutat él, t_1-ből pedig p_0-ba, mert több tokent termel mint amennyit elvesz. A gráfban egy forrás SCC lesz, amelyet p_0 és t_1 alkot. t_1 tüzeléséhez elég egy token p_0-ba, ahova a t_2 tud termelni. A*

növelő megkötés tehát $|t_2| \geq 1$ alakú lesz. A megkötés alkalmazása után az $x' = (1, 1, 1, 1)$ új megoldásvektort kapjuk, amely t_2, t_1, t_0, t_3 sorrendben realizálható.



3.4. ábra. Növelő megkötés példa

1. Tétel. *(Megoldások elérhetősége).* Amennyiben az elérhetőségi problémának van megoldása, az állapotegyenlet egy realizálható megoldását el tudjuk érni úgy, hogy folyamatosan adunk hozzá megkötéseket, az ugró megkötéseket mindig átalakítva, mielőtt növelőt adunk hozzá.

Bizonyítás. Az állítás az 1. lemmából következik. □

2. Tétel. *(Az algoritmus mindig eléri a megoldásokat).* Az előbbi tétel akkor is igaz, ha a növelő megkötéseket csak a 6. lemma segítségével készítjük, az ugró megkötések pedig mind $|t_i| < x(t_i)$ alakúak, ahol x a megoldásvektor.

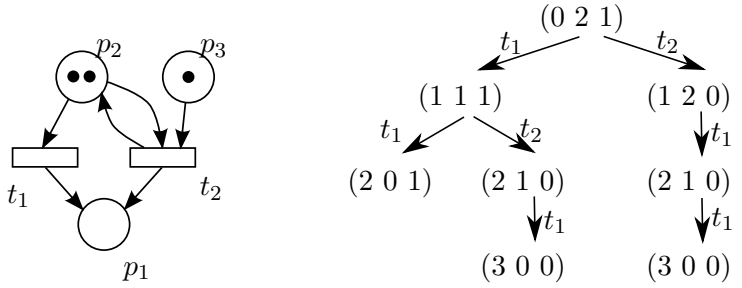
Bizonyítás. Legyen σ egy realizálható tüzelési sorozat úgy, hogy $\wp(\sigma) = x$ és b egy minimális megoldás úgy, hogy $b \leq x$. A 2. lemma alapján b-t megkaphatjuk ugró megkötések segítségével. Ha b realizálható, kész vagyunk. Amennyiben b nem realizálható, akkor a 4., 5. és 6. lemmák segítségével növelő megkötést generálhatunk hozzá, amely tokeneket próbál termelni a szükséges helyekre úgy, hogy közben nem veszik el realizálható megoldás. Legyen a növelő megkötéssel kiegészített rendszer megoldása y, ekkor $b \leq y$ teljesül. Ha y realizálható, akkor kész vagyunk. Ha $y \leq x$, akkor újból növelő megkötést adunk hozzá, ellenkező esetben ugró megkötésekkel előállítjuk az $y_1, \ldots, y_n$ megoldásokat, amelyek kielégítik a növelő megkötést, de nem összehasonlíthatók y-al. Mivel nem veszett el teljes megoldás, ezért valamilyen k-ra $y_k \leq x$ kell, hogy teljesüljön. Ekkor y_k-val folytatjuk. Ha nem realizálható, akkor növelő megkötéseket adunk hozzá, amíg szükséges. Mivel x véges és a megkötések által a megoldásvektorok monoton növekednek, ezért vagy találunk útközben egy másik realizálható megoldást, vagy elérjük x-et. □

A 2. tétel azt mondja ki, hogy amennyiben létezik megoldás, azt az algoritmus meg is találja. Dolgozatunkban elméleti eredményként bemutatjuk (lásd 3.5. fejezet), hogy ez a tétel nem igaz. Konstruktív ellenpéldát mutatunk a teljességre és a helyességre is, azaz mutatunk hálókat, amelyeknél ismerjük a megoldást, de az algoritmus nem tudja eldönteni, illetve helytelen választ ad.

3.2.4. Részleges megoldások generálása

Egy x megoldásvektorhoz (és a hozzá tartozó $\mathcal{C}$ megkötés-halmazhoz) könnyen találhatunk részleges megoldásokat. Egy olyan fát kell felépíteni és bejárni, ahol a csúcsok tokeneloszlások, az élekhez pedig tranzíciók vannak rendelve. Egy m_1 és m_2 csúcs között akkor fut t él, ha $m_1[t\rangle m_2$ teljesül. A fában minden gyökérből levélbe vezető úton a $t_i \in T$ tranzíció legfeljebb $x(t_i)$-szer szerepelhet. Minden levélbe vezető út egy maximális tüzelési sorozatot reprezentál, amihez egy új $(\mathcal{C}, x, \sigma, r)$ részleges megoldás tartozik.

Mélységi kereséssel bejárhatjuk a teljes fát anélkül, hogy az egész fel lenne építve egyszerre. Ennek ellenére az ilyen naív módon bejárt fa mérete exponenciálisan nőhet az x megoldásvektor méretéhez képest. A következő fejezetben többek között olyan optimalizációkat is bemutatunk, amelyek ezen igyekeznek javítani.



(a) Petri-háló (b) Részleges megoldás fa

3.5. ábra. Tegyük fel, hogy a 3.5(a). ábrán látható Petri-háló esetén a (2 1) megoldásvektort kaptuk. Ekkor a részleges megoldás fa a 3.5(b). ábrán látható módon néz ki.

3.2.5. Optimalizációk

Ebben a fejezetben bemutatjuk az algoritmus hatékonyságát növelő optimalizációkat.

Stubborn set

Amikor egy x megoldásvektorhoz keresünk részleges megoldásokat, a bejárandó fa mérete x-ben lévő tranzíciók számával exponenciális arányban nőhet, így igen hamar kezelhetetlen méretűvé válhat. Ennek a fának a méretét tudjuk csökkenteni az úgynevezett részleges rendezést használó algoritmusok segítségével. Mi munkánk során a [10] alapján a stubborn set elnevezésű részleges rendezés algoritmusát használtuk. Az állapottér robbanás (state explosion) probléma enyhítésére számos módszert ismert, ezek egyike az úgynevezett *stubborn set* módszer. A részleges rendezést használó algoritmusok alapelve, hogy particionálják a tranzíciókat fontosságuk alapján: a stubborn set módszere a mindenképpen eltüzelendő tranzíciókat az úgynevezett stubborn set halmazba teszi. Különböző tulajdonságokhoz különböző stubborn set halmaz definíciókat fejlesztettek ki. Az eljárás hasonló módon működik, mint a teljes állapottér generálása, az egyetlen különbség az, hogy amikor elérünk egy állapotot, akkor tranzíciók egy halmazát, úgynevezett stubborn setet rendelünk hozzá. Ezután csak a stubborn setben lévő tranzíciókon (ezek közül is csak az engedélyezetteken) haladunk tovább. Esetünkben egy elért állapot esetén elég úgy meghatároznunk a hozzá tartozó stubborn set halmazt, mintha csak azok a tranzíciók lennének a Petri-hálóban, amik még benne vannak az r vektorban. Hiszen ha kiveszünk egy tranzíciót, amiről tudjuk, hogy sose fogjuk eltüzelni, az nem változtatja meg a jelenlegi állapotból bejárható utakat.

Nekünk az a célunk, hogy az x megoldásvektort teljesen, vagy részlegesen eltüzeljük, amíg el nem akadunk. Ehhez elég, ha a stubborn set halmazok az alábbi **D1** és **D2** tulajdonságokat kielégítik:

- **D1** Ha $t \in stub(M_0)$, $t_1, ..., t_2 \notin stub(M_0)$, $M_0[t_1 t_2, ..., t_n\rangle M_n$, és $M_n[t\rangle to M'_n$, akkor létezik M'_0 állapot, hogy $M_0[t\rangle M'_0$ és $M'_0[t_1 t_2 ... t_n\rangle M'_n$.
- **D2** Ha M_0-nak van engedélyezett tranzíciója, akkor van legalább egy $t_k \in stub(M_0)$, amelyre teljesül: ha $t_1, ..., t_2 \notin stub(M_0)$ és $M_0[t_1 t_2, ..., t_n\rangle M_n$, akkor M_n állapot-

ban t_k engedélyezve van. Minden ilyen tulajdonságú tranzíciót a stubborn set egy kulcstranzíciójának nevezünk.

Ha egy x megoldásvektorhoz (és a hozzá tartozó $\mathcal{C}$ megkötés-halmazhoz) a keresőfa teljes felépítésével találunk egy $(\mathcal{C}, x, \sigma, r)$ részleges megoldást, akkor a **D1** és **D2** tulajdonságot kielégítő stubborn set-ek használatával is kapunk egy $(\mathcal{C}, x, \sigma', r)$ részleges megoldást (azaz csak a tüzelési sorrendben tér el). Az algoritmus természetéből adódóan, ebből ugyanúgy indulunk tovább, mint a $(\mathcal{C}, x, \sigma, r)$ részleges megoldásból, tehát a stubborn set algoritmus használatával nem veszítünk el számunkra hasznos parciális megoldásokat.

Bizonyítás. Indirekt úton tegyük fel, hogy létezik $(\mathcal{C}, x, \sigma, r)$ parciális megoldás, és stubborn set halmazok segítségével nem kaphatunk $(\mathcal{C}, x, \sigma', r)$ csak tüzelési sorrendben eltérő megoldást. Ennek szükséges feltétele, hogy ha $\sigma = t_1 t_2 ... t_n$ és $M_0[t_1\rangle M_1[t_2\rangle M_2[t_3\rangle ...[t_n\rangle M_n$, akkor létezik olyan $M_d \in M_0, M_1 ... M_n$, hogy M_d-ből teljes faépítéssel kaphatunk még $(\mathcal{C}, x, \sigma', r)$ típusú részleges megoldást, de ha tovább lépünk egy $t \in stub(M_d)$ tranzícióval egy $M_d[t\rangle M'_d$ csúcsba, akkor onnan már nem kapható $(\mathcal{C}, x, \sigma', r)$ típusú részleges megoldás. Mivel M_0-ból még tudjuk hogy elérhető volt, tehát valahol a bejárás közben rontották el a stubborn set halmazok.

Tehát $M_d[t_{d+1} t_{d+2} t_d ... t_n\rangle M_n$ úton megkapunk egy $(\mathcal{C}, x, \sigma', r)$ típusú megoldást, és M_d-ig eljutunk stubborn setek halmazokkal is ($M_d = M_0$ lehetséges).

- Ha $t_{d+1}, t_{d+2}, t_d, ..., t_n$ tranzíciók közül egy sem eleme $stub(M_d)$-nek, akkor **D2** tulajdonság miatt M_n-ben van még egy tüzelhető tranzíció, mivel $stub(M_d)$-ben volt egy kulcstranzíció, ami még tüzelhető. Tehát M_n nem egy levél, így nem készíthettünk belőle részleges megoldást. Azaz ez az eset nem állhat fenn.

- Ha $t_{d+1}, t_{d+2}, t_d, ..., t_n$ tranzíciók közül vannak, melyek elemei $stub(M_d)$-nek, akkor vegyük ezek közül a legkisebb indexű elemet: t_k-t. Ekkor $t_d t_{d+1} ... t_{k-1} \notin stub(M_d)$. Ekkor **D1** miatt $t_k t_d t_{d+1} ... t_{k-1}$ sorozat is tüzelhető M_d-ből, tehát $M_d[t_k t_d t_{d+1} ... t_{k-1} t_{k+1} ... t_n\rangle M_n$, azaz M_d-ből $t_k \in stub(M_d)$ tranzícióval tovább haladva se válik M_n elérhetetlenné, tehát ez az eset sem állhat fenn.

□

Ahhoz, hogy ezt az algoritmusunkban fel tudjuk használni, szükségünk van egy olyan módszerre, amivel egy adott állapothoz hatékonyan meg tudjuk határozni a **D1** és **D2** tulajdonságú stubborn set halmazokat. Minél kisebb a stubborn set elemszáma, annál hatékonyabb lesz az algoritmusunk, mivel kevesebb trajektóriát fogunk bejárni. Viszont a meghatározásának nem szabad túl időigényesnek lennie. Több módszer ismert, mi az alábbi definíciót használtuk fel: $T_s \subseteq T$ halmaz a **D1** és **D2** tulajdonságokat kielégítő, M állapothoz tartozó stubborn set, ha:

- Ha $t \in T_s$ és M-ből nem engedélyezett t , akkor létezik $p \in \bullet t$, melyre $M(p) < w^-(p,t)$ és $\{t' \mid W^-(p,t') < W^+(p,t') \wedge W^-(p,t') \leq M(p)\} \subseteq T_s$.

- Ha $t \in T_s$ és M-ből engedélyezett t , akkor minden $p \in \bullet t$ esetén
  $\{t' \mid min(W^+(p,t), W^+(p,t')) < min(W^-(p,t'), W^-(p,t))\} \subseteq T_s$ vagy
  $\{t' \mid min(W^+(p,t), W^-(p,t')) < min(W^+(p,t'), W^-(p,t))\} \subseteq T_s$.

- Létezik legalább egy $t_k \in T_s$, amire M-ből engedélyezett t_k , és minden $p \in \bullet t_k$, helyre: $\{t' \mid W^+(p,t') < min(W^-(p,t'), W^-(p,t_k))\} \subseteq T_s$.

Ennél vannak egyszerűbb definíciók, amik nem ennyire szigorúak, ezért gyorsabban, de nagyobb stubborn set halmazokat határoznak meg. Mi ezzel a definícióval is még elfogadható sebességgel tudunk dolgozni. A 3 algoritmussal dolgozunk, ahol a fenti három feltételre **F1**,**F2** és **F3** néven hivatkozunk.

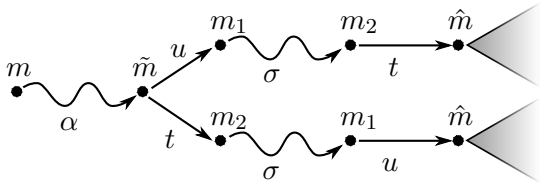
Algoritmus 3: Stubborn set meghatározása adott állapothoz

bemenet: (N, m) petri háló; r tüzelésre váró tranzíciók halmaza
kimenet : T_s stubborn set

```
1  T_s = ∅;
2  t_k = r(0);
3  tmp = ∞;
4  for minden t_k ∈ r do
5  |  T_tmp = {t_k};
6  |  for minden p ∈ •t_k és minden t' ∈ T ∧ m[t'⟩ do
7  |  |  if t' ∉ T_tmp ∧ W+(p,t') < min(W−(p,t'), W−(p,t_k)) then
   |  |  T_tmp = T_tmp ∪ {t'} ;
8  |  end
9  |  if |T_tmp| < tmp then tmp = |T_tmp| ;
10 |  t_k = t' ;
11 end
12 T_s = {t_k} ;
13 while ∃t ∈ T_s amire F1 vagy F2 nem teljesül do
14 |  T_s = T_s ∪ {legkevesebb tranzíció, amivel T_s-t kiegészítve F1 és F2 teljesül t-re};
15 end
```

Állapotalapú részleges rendezés

Megfigyelhető, hogy amikor egy tranzíciónak többször is el kell tüzelnie ($x(t) > 1$), a stubborn set módszer önmagában nem elég hatékony. Gyakran előfordul, hogy ugyanazt az állapotot több, csak a tranzíciók sorrendjében különböző tüzelési sorozattal érjük el. A 3.6. ábra mutat egy példát, ahol egy α tüzelési sorozat után az $u\sigma t$ és $t\sigma u$ sorozatok is ugyanahhoz az $\tilde{m}$ állapothoz vezetnek úgy, hogy csak a tranzíciók sorrendjében különböznek. Ekkor ha már az $u\sigma t$ utáni részfát bejártuk, akkor fölösleges a $t\sigma u$ utánit is, ugyanis a végén kialakuló részleges megoldások szempontjából mindegy a tranzíciók sorrendje. Az ugró és növelő megkötésekben sehol nem használjuk ki sem a tranzíciók sorrendjét, sem a köztes állapotokat.



3.6. ábra. Ha $\alpha t\sigma u$ és $\alpha u\sigma t$ is eltüzelhető, akkor az $\hat{m}$ utáni részfák megegyeznek, tehát elég az egyiket feldolgozni.

T-invariáns alapú szűrés

Miután egy $\mathrm{ps} = (\mathcal{C}, x, \sigma, r)$ részleges megoldás felhasználásakor hozzáveszünk egy y invariánst x megoldásvektorhoz, előfordulhat, hogy az $x + y$ megoldásvektorból egy $(\mathcal{C}', x + y, \sigma', r)$ részleges megoldást kapunk. Azaz a hozzávett invariánst teljesen eltüzeltük, de ugyanazok a tranzíciók maradtak a maradékvektorban. Mivel az algoritmus a részleges megoldások feldolgozásánál csak az elért állapottal és a maradékvektorral dolgozik, ezért ez az új részleges megoldás semmilyen információt sem hordoz a számunkra,

tehát felesleges vele tovább haladnunk. Viszont lehetséges, hogy az invariánssal jó irányba haladtunk, csak alulbecsültük a szükséges tokenek számát. Ebben az esetben σ' tüzelés folyamán közelebb kerülhetünk egy $t \in r$ tranzíció eltüzeléséhez (a közelebb kerülés alatt azt értjük, hogy t eltüzeléséhez kevesebb token hiányzik, mint a végállapotban). Ekkor ebből a köztes állapotból érdemes tovább indulnunk. Tehát ha egy $(\mathcal{C}', x + y, \sigma', r)$ részleges megoldást kapunk (ahol y egy invariáns) és már volt egy $(\mathcal{C}, x, \sigma, r)$ részleges megoldásunk, ahol $\mathcal{C}$ és $\mathcal{C}'$ csak növelő megkötésekben tér el, és $(\mathcal{C}', x + y, \sigma', r)$ állapotait visszanézve nem találunk javulást, akkor ezt a megoldást eldobhatjuk. Amennyiben egy köztes állapotból folytatjuk, a maradékvektorban megjelennek olyan tranzíciók, amelyek el tudnának tüzelni, de szándékosan nem tüzeljük el. Növelő megkötés keresésekor ezeket meg kell különböztetni azoktól a tranzícióktól amelyek eddig sem voltak eltüzelhetőek.

Részleges megoldások tárolása

Ugyanazt a részleges megoldást többször is megkaphatjuk, ilyenkor ha már egyszer feldolgoztuk, akkor ezen az ágon a keresést már elvégeztük, a második alkalommal ugyanezt az eredményt kapnánk: szűrni kell az ilyen eseteket. Kérdés, hogy mikor tekinthetünk két részleges megoldást egyenlőnek. Egy részleges megoldásnál a tüzelési sorozatot csak a T-invariáns alapú szűrés optimalizációnál használjuk. Ez az optimalizáció viszont bizonyos problémákkal küzd, és az erre alkalmazott javításokat később ismertetjük. Ezen javítások után már itt sem számít a tüzelési sorrend. Azaz azt mondhatjuk, hogy ha van egy $(\mathcal{C}, x, \sigma, r)$ és egy $(\mathcal{C}', x', \sigma', r')$ részleges megoldásunk, ahol $x = x' \wedge r = r' \wedge \mathcal{C} = \mathcal{C}'$ akkor elég csak az egyikkel foglalkoznunk, a másikat eldobhatjuk, hiszen az algoritmus azonos módon halad tovább belőlük. Az $x = x' \wedge r = r'$ feltételek ellenőrzése triviális, egyszerűen eldönthető. Ennél a pontnál fontos megemlíteni, hogy amikor megvizsgáljuk, hogy két megkötés halmaz ekvivalens-e, és azt mondjuk, hogy nem, pedig azok, akkor nem történik baj, csak feleslegesen feldolgozzuk a részleges megoldást. Fordítva viszont nem hibázhatunk, mivel ekkor értékes eredményeket, azaz megoldásokat is elveszíthetünk. Ha a megkötés-halmazokat tekintjük, akkor fogunk két megkötés halmazt (lineáris egyenlőtlenség-rendszert) ekvivalensnek tekinteni, ha ugyanazokat a megkötéseket tartalmazzák. Tehát a módszer az, hogy minden megkötés-halmazból kiszűrjük a felesleges megkötéseket. Felesleges megkötés alatt a megkötés-halmaz maradék részéből következő megkötéseket értjük. Erre egy példa az alábbi. Vegyünk egy lineáris egyenlőtlenség-rendszert, melynek két változója van: x és y. Tartalmazzon 5 egyenlőtlenséget: $x \geq 0$, $y \geq 0$, $x+y \leq 5$, $y - 0.1x \leq 10$ és $x \leq 10$. Itt az első háromból következik az utolsó két megkötés, tehát azokat eldobhatjuk anélkül, hogy a reprezentált halmaz (esetünkben a sokszög belső pontjai) megváltozna.

Ha adott egy $\mathcal{C}$ lineáris egyenlőtlenség-rendszer, akkor abból oly módon szűrhetjük ki a felesleges egyenlőtlenségeket, hogy sorra végigmegyünk mindegyiken és megnézzük, hogy következik-e a többiből. Amennyiben igen, akkor kidobjuk. $c \in \mathcal{C}$ egyenlőtlenségre $\mathcal{C} \setminus c \Rightarrow c$ (azaz $\mathcal{C} \setminus c$-ből következik c) ekvivalens azzal, hogy $(\mathcal{C} \setminus c) \cup (\neg c)$ megoldható. Az utóbbi egy LP probléma, amit meg tudunk oldani. Egy egyenlőtlenséget egy olyan egész számokat tartalmazó vektorral reprezentálunk, melynek első $|T|$ eleme ($|T|$-vel jelöljük a tranzíciók számát) az egyes változók együtthatóit tartalmazza, az ezt követő a relációs jelet ($\leq \geq =$) kódolja, az ezt követő pedig a jobb oldalon álló konstanst. Az így kapott vektorokat lexikografikus sorrendben tároljuk. Ekkor két megkötés halmaz egyezését a megkötések számával lineáris időben tudunk ellenőrizni, sőt a kisebb és nagyobb relációkat is tudjuk köztük definiálni, melynek eldöntése szintén lineáris időben megtehető. Ezek után két részleges megoldás között is tudunk kisebb-nagyobb relációt definiálni, mégpedig oly módon, hogy először a megoldás vektort (lexikografikus sorrend) vizsgáljuk, majd ha az megegyezik, akkor a maradék vektorokat (szintén lexikografikus sorrendben)

hasonlítjuk össze. Ha az is megegyezik, akkor a megkötés halmazaikat hasonlítjuk a fent bemutatott módon. Így, ha eddig n részleges megoldásunk volt és ezek átlagosan c megkötést tartalmaznak, akkor $o(c \times |T| \times log(n))$ időben meg tudjuk nézni, hogy egy újabb részleges megoldás szerepelt-e már.

3.3. Algoritmikus hiányosságok és fejlesztések

Ebben a fejezetben bemutatjuk az algoritmus vizsgálata során felfedezett hiányosságokat, illetve az általunk kifejlesztett megoldásokat ezekre a problémákra.

3.3.1. Rész-elérhetőségi probléma

A háttérismeretek között már foglalkoztunk a rész-elérhetőségi problémával (2.2. fejezetben) és annak hasznosságával. Az általunk vizsgált algoritmus csak konkrét állapotok elérhetőségének vizsgálatára képes. Kiterjesztett algoritmusunkban megvalósítottuk a rész-elérhetőségi probléma vizsgálatának lehetőségét, amely új algoritmust ebben a fejezetben mutatunk be. A predikátumokat a háttérismeretek (2.2.) fejezetben már bemutatott $Am \geq b$ módon definiálhatjuk, ahol A együtthatómátrix, b együtthatóvektor, m pedig az állapot, amire a feltételeknek teljesülnie kell.

Ahhoz, hogy a predikátumokat az állapotegyenlet segítségével kezelni tudjuk, a helyekre vonatkozó feltételt át kell kell alakítani tranzíciókra vonatkozóvá.

A predikátum

$$Am \geq b \tag{3.2}$$

alakjába az elért m állapot helyére behelyettesíthetjük az állapotegyenletet:

$$m_0 + Cx = m' \tag{3.3}$$

így a kapott egyenlőtlenség

$$\begin{array}{rcl} A(m_0 + Cx) & \geq & b \\ (AC)x & \geq & b - Am_0 \end{array} \tag{3.4}$$

alakú, amelyben egyedül a tranzíciók tüzelési számai (x vektor) az ismeretlenek.

Az algoritmust úgy kell módosítani, hogy az állapotegyenlet helyett ezt a lineáris egyenlőtlenség-rendszert használja kiindulásként és ehhez adja hozzá a további (ugró vagy növelő) megkötéseket.

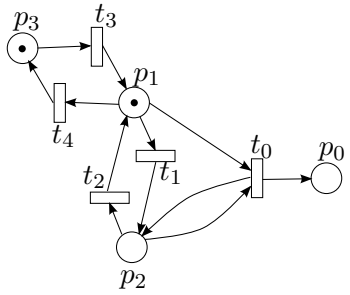
3.3.2. Javulás szigorú definíciója

Mielőtt egy részleges megoldást kiszűrünk, a tüzelési sorozatában visszalépésekkel megkeressük azokat az állapotokat, amelyekből új részleges megoldást készítve érdemes lenne tovább haladni. Az általunk vizsgált algoritmus [17] akkor tekint egy állapotot a végállapotnál jobbnak, ha valamely tüzelni nem tudó tranzíció bemenő helyein összesen több token van (mint a végállapotban). Ezzel a módszerrel azonban olyan állapotokat veszíthetünk el, ahol összességében ugyan kevesebb token van az adott tranzíció bemeneti helyein, mint a végállapotban, viszont lehetnek olyan helyek ahol lokálisan több token található. Amennyiben nem vesszük észre az ilyen állapotokat, akkor teljes megoldások veszhetnek el.

Példának vegyük a 3.7. ábrán látható Petri-hálót a $(0, 1, 0, 1) \rightarrow (1, 0, 0, 1)$ elérhetőségi problémával. Minimális megoldásvektor az $x_0 = (1, 0, 0, 0, 0)$, azaz t_0 tranzíció eltüzelése. Mivel t_0 nem engedélyezett p_2 miatt, egyetlen $PS_0 = (\emptyset, x, \sigma_0 = (), r_0 = (1, 0, 0, 0, 0))$

részleges megoldást találunk. Növelő megkötés keresésekor az algoritmus a t_1 tranzíciót találja meg, amely tokent tud termelni p_2 helyre. Ekkor a $|t_1| \geq 1$ megkötés következtében az új x_1 megoldásvektor kiegészül a t_1, t_2 invariánssal. Ehhez egyetlen $PS_1 = (\{|t_1| \geq 1\}, x_1, \sigma_1 = (t_1, t_2), r_1 = (1, 0, 0, 0, 0))$ részleges megoldás tartozik, melyben a maximális tüzelési sorozatokra törekvés miatt a t_1 és t_2 tranzíciót is eltüzeljük. PS_1 csak egy eltüzelt T-invariánsban különbözik PS_0-tól, tehát kiszűrhető. A tüzelési sorozatban visszalépésekkel egyedül a $(0, 0, 1, 1)$ állapotot találjuk, amelyben a $(0, 1, 0, 1)$ végállapothoz hasonlóan összesen 1 token van t_0 bemenő helyein. Az eredeti algoritmus ezt a köztes állapotot nem folytatná, és mivel nincs más részleges megoldás, sem megoldásvektor, az elérhetőséget nem tudja eldönteni.

Amennyiben figyelembe vennénk azt, hogy a p_2 hely szempontjából a $(0, 0, 1, 1)$ állapot jobb mint a végállapot és innen egy $PS_2 = (\{|t_1| \geq 1\}, x_2 = x_1, \sigma_2 = (t_1), r_2 = (1, 0, 1, 0, 0))$ részleges megoldással folytatnánk, akkor t_0 miatt p_1 helyre kellene tokent keresni. Ide a t_3 és a t_2 tranzíció tud tokent termelni, de mivel t_2-t nem tüzeltük el, azaz benne van a maradékvektorban, ezért az algoritmus a t_3, t_4 invariánst veszi hozzá a megoldáshoz. Az új x_3 megoldásvektorban minden tranzíció pontosan egyszer szerepel, amelyhez a $\sigma_3 = (t_1, t_3, t_0, t_2, t_4)$ tüzelési sorozattal realizálható teljes megoldás tartozik.



3.7. ábra. Az ábrán látható háló $(0, 1, 0, 1) \to (1, 0, 0, 1)$ elérhetőségi problémája esetén, ha csak szigorúan több token esetén folytatjuk egy köztes állapotból, akkor elveszítünk egy teljes megoldást.

Az eredeti algoritmus formálisan akkor tekinti a tüzelési sorozat egy m_i állapotát jobbnak az m' végállapotnál, ha létezik olyan $t \in T, r(t) > 0$ tranzíció, amire

$$\sum_{p_j \in \bullet t \wedge m'(p_j) < W(p_j,t)} m_i(p_j) > \sum_{p_j \in \bullet t \wedge m'(p_j) < W(p_j,t)} m'(p_j) \tag{3.5}$$

teljesül, azaz t azon bemenő helyein, ami miatt nem engedélyezett, több token van m_i állapotban mint m'-ben.

Ezen módszer előnye, hogy gyorsan számolható, de ahogy a 3.7. ábra Petri-hálója is mutatja, ez a feltétel túl szigorú.

Általános feltétel a javulásra

Az algoritmus továbbfejlesztésében az m_i állapotot akkor tekintjük jobbnak, ha létezik olyan $t \in T, r(t) > 0$ tranzíció és $p \in \bullet t$ hely, amire

$$m'(p) < W(p, t) \wedge m_i(p) > m'(p) \tag{3.6}$$

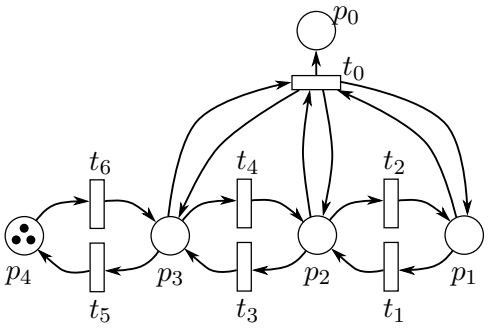
teljesül, azaz ha legalább egy $p \in \bullet t$ helyen, ami miatt nem engedélyezett t, több token van, mint a végállapotban, akkor ezt jobb állapotnak tekintjük.

Az általunk definiált (3.6.) feltétel minden olyan m_i állapotra teljesül, amire (3.5.) igaz, ugyanis ha nincs olyan $p \in \bullet t$ hely, amire $m_i(p) > m'(p)$ teljesülne, az azt jelentené, hogy minden $p \in \bullet t$ helyen kevesebb token van, mint a végállapotban, tehát az összegük is kevesebb. Ezzel a definícióval tehát az eredeti algoritmushoz képest a Petri-hálók egy bővebb részhalmazára tudjuk eldönteni az elérhetőséget, többek között a 3.7 hálóra is.

3.3.3. Állapotalapú részleges rendezés és a T-invariáns alapú szűrés

Az állapotalapú részleges rendezés során kihasználjuk azt, hogy az alap algoritmus szempontjából mindegy a tranzíciók sorrendje. A különböző sorrendek közül csak az elsőt tartjuk meg. A T-invariáns alapú szűrésnél ez már nem használhatjuk, mert a visszalépegetés során megvizsgáljuk a köztes állapotokat is, hogy van-e javulás bizonyos tranzíciók szempontjából.

Példaként tekintsük a 3.8. ábrán látható Petri-hálót és a $(0,0,0,0,3) \to (1,0,0,0,3)$ elérhetőségi problémát. A megoldás nyilvánvalóan az, hogy t_6 háromszori, t_4 kétszeri és t_2 egyszeri eltüzelésével a p_1, p_2, p_3 helyek mindegyikére eljuttatunk egy-egy tokent. Ekkor t_0 eltüzelhető, majd a tokeneket a t_1, t_3, t_5 tranzíciók segítségével visszajuttathatjuk p_4-be.



3.8. ábra. T-invariáns alapú szűrésnél a tranzíciók sorrendje is számít.

Az algoritmus kezdetben a t_0 eltüzelését adja minimális megoldásként, ez azonban a p_1, p_2, p_3 helyek miatt nem engedélyezett. Növelő megkötés keresésekor ez a 3 hely mind egy-egy forrás SCC lesz, ahova 1-1 token szükséges. A 3 hely külön SCC-ben van ezért az algoritmus kezdetben egy token körbevitelével próbálkozik. Az algoritmus lefutásából most csak a lényeges részeket emeljük ki, néhány köztes megoldásvektort, illetve biztosan zsákutcát jelentő részleges megoldást nem említünk meg.

Elsőként a $t_1, t_2, \ldots, t_6$ T-invariáns segítségével eléri az algoritmus, hogy a p_1, p_2, p_3 helyeken körbemenjen egy p_4-ből kivett token, de mivel egyszerre csak a helyek egyikén van ez a token, ezért t_0 nem tud tüzelni és a token visszakerül p_4-be. A részleges megoldások T-invariáns alapú szűrése nélkül végtelen ciklusba kerülnénk, ugyanis ebben az állapotban ismét a kezdőállapotnak megfelelő feltételek állnak fenn, így újból a $t_1, t_2, \ldots, t_6$ invariánst venné hozzá az algoritmus.

A részleges megoldások szűrésével ezt a végtelen ciklust elkerüljük és a visszalépések során a végállapothoz képest p_0 szempontjából jobb állapotok között szerepelni fog az a három állapot, amikor a token p_1, p_2, p_3 helyek valamelyikén van. Ezek közül bármelyikkel is folytatjuk, az algoritmus rájön, hogy még egy tokent körbe kell vinni, ezért az új megoldásvektorban t_5, t_6 invariáns már kétszer is szerepelni fog (t_1, t_2, t_3, t_4 egyszer).

Ekkor jön ki a különbség a tüzelési sorrendek között:

- $t_6, t_6, t_4, t_2, t_1, t_3, t_5, t_5$ sorrendben eltüzelve a tranzíciókat először két tokent teszünk p_3-ra, majd az egyiket elvisszük p_1-ig, aztán mindkettőt vissza p_4-be. A végállapot csak egy T-invariánsban különbözik a minimális megoldástól (t_0 tüzelése), de a visszalépések során megtaláljuk azt a javulást jelentő állapotot, amikor p_3-on 2 token van és innen folytatva, az algoritmus megtalálhatja a teljes megoldást.

- Ha viszont $t_6, t_4, t_2, t_1, t_3, t_5, t_6, t_5$ sorrendben tüzeljük el a tranzíciókat, akkor előbb az első tokent visszük el p_1-ig és vissza, majd a második tokent p_3-ba és vissza. Mivel a két token egyszerre nem volt p_1, p_2, p_3 helyeken, ezért a visszalépések során

nem találunk olyan állapotot, ahol az eddig elért egy tokennél több lenne t_0 bemenő helyein, így az algoritmus nem találja meg a teljes megoldást.

A Petri-hálók nem-determinisztikus működése miatt nem tudhatjuk, hogy melyik lesz az a sorrend, amit később kapunk meg és emiatt eldobunk. Erre bizonyíték az, hogy ugyanazokat a tranzíciókat más sorrendben adtuk meg az eredeti algoritmust implementáló programban [3] és az egyik esetben megtalálta a megoldást, a másik esetben pedig nem csak hogy nem tudta eldönteni a problémát, hanem hibásan a „nem elérhető” eredményt adta.

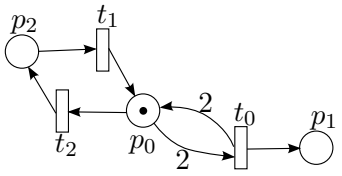
Ezt a problémát megoldhatjuk úgy, hogy amikor egy részleges megoldást feleslegesnek ítélünk meg, akkor a megoldásvektort újra eltüzeljük, de az állapotalapú részleges rendezés optimalizáció nélkül, így már nem vesznek el részleges megoldások. Az itt keletkező köztes állapotokra pedig megvizsgáljuk, hogy történt-e javulás. Amennyiben igen, akkor ezt a javulást felhasználva, ebből az állapotból kell tovább haladnunk. Azért jobb ez a módszer annál, mintha abszolút nem használnánk az állapotalapú részleges rendezés optimalizációt, mert így csak akkor kell az egész fát bejárnunk, ha kiszűrésre került egy megoldás.

3.3.4. Rész-elérhetőségi probléma vizsgálata növelő megkötés keresésekor

Növelő megkötések generálásakor először megkeressük a helyeket, ahova tokent kell termelni, majd megbecsüljük a szükséges tokenek számát. A harmadik lépésben olyan tranzíciókat próbálunk belevenni a megoldásba, amelyek kielégíthetik ezt a szükséges tokenszámot. Ezt akár egy rész-elérhetőségi problémaként is felfoghatjuk: helyek egy $P_i \subseteq P$ részhalmazára kell összesen legalább n tokent termelni. Ekkor leellenőrizhetnénk az állapotegyenlet segítségével, hogy a kezdőállapotból elérhető-e egyáltalán olyan állapot, ahol P_i helyein legalább n token van. Ezzel hatékonyan vághatnánk a keresési teret, ugyanis előfordulhat, hogy növelő megkötések hozzávételével csak sokkal később, vagy akár soha nem jönne rá az algoritmus, hogy az adott helyre nem is lehet elég tokent termelni.

Tekintsük a 3.9. ábrán látható Petri-hálót az $(1,0,0) \to (1,1,0)$ elérhetőségi problémával. A háló szomszédossági mátrixa:

$$C = \begin{pmatrix} 0 & 1 & -1 \\ 1 & 0 & 0 \\ 0 & -1 & 1 \end{pmatrix} \tag{3.7}$$



3.9. ábra. Rész-elérhetőségi probléma vizsgálata növelő megkötés keresésekor

Az algoritmus egyetlen minimális megoldásként az $x = (1,0,0)$ megoldásvektort találja meg, amely kielégíti az állapotegyenletet. Ehhez egy részleges megoldás tartozik, mert a kívánt t_0 tranzíciót nem tudjuk eltüzelni. Növelő megkötés keresésekor rájövünk, hogy t_0 engedélyezéséhez p_0-ban 2 token szükséges, azaz még egyet termelni kell. A maradékvektorban nem szereplő tranzíciók közül erre a t_1 egyszeri tüzelése alkalmas, tehát a kapott növelő megkötés $|t_1| \geq 1$ alakú. Az új egyenlőtlenség-rendszer minimális megoldása az $x' = (1,1,1)$ vektor. Ehhez egy részleges megoldás tartozik t_1, t_2 tüzelési sorozattal és $(1,0,0)$ maradékvektorral. Ekkor újra a p_0 helyre szeretnénk még egy tokent termelni, amihez ismét a t_1 tranzíciót tüzeltetjük, immár $|t_1| \geq 2$ megkötéssel. Amennyiben nem

szűrjük T-invariáns alapon a megoldásokat, végtelen ciklusba kerülünk, mert az algoritmus mindig a t_1, t_2 invariánst veszi hozzá és tüzeli el, de a t_0 sose lesz engedélyezett.

A megoldások szűrésével észrevehetjük, hogy csak egy T-invariánsban térünk el az előző megoldástól és mivel a tüzelési sorozatban visszalépkedve sincs t_0 szempontjából jobb állapot, elvethetjük ezt az ágat. Ekkor azonban a szűrés miatt csak annyit mondhat ki az algoritmus, hogy nem tudja eldönteni a problémát.

A növelő megkötések keresésekor azonban felírhatunk egy olyan szükséges feltételt, hogy a kiinduló $m_0 = (1, 0, 0)$ állapotból léteznie kell, olyan m_1 állapotnak, ahol p_0-n legalább 2, a többi helyen pedig nemnegatív számú token van. Ez formálisan felírva a következőhöz vezet:

$$m_0 + Cx \geq m_1 \tag{3.8}$$

$$\begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 & 1 & -1 \\ 1 & 0 & 0 \\ 0 & -1 & 1 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} \geq \begin{pmatrix} 2 \\ 0 \\ 0 \end{pmatrix} \tag{3.9}$$

Átrendezve és lineáris egyenlőtlenség-rendszerként felírva:

$$x_1 - x_2 \geq 1 \tag{3.10}$$

$$x_0 \geq 0 \tag{3.11}$$

$$-x_1 + x_2 \geq 0 \tag{3.12}$$

Ekkor a 3.10. és 3.12. összeadásával a $0 \geq 1$ egyenlőtlenséget kapjuk, ami nyilvánvalóan nem igaz. Ezzel bebizonyítottuk, hogy m_0-ból nem érhető el olyan állapot, ahol p_0-n legalább 2 token van, ezáltal ezt a részleges megoldást tényleg eldobhatjuk. A T-invariáns alapú szűréssel ellentétben itt nem veszíthetünk el teljes megoldást, tehát erre a hálóra kimondhatjuk, hogy a célállapot nem elérhető. A továbbfejlesztett algoritmusban ezt az ötletet általánosítottuk és valósítottuk meg.

Tegyük fel, hogy a növelő megkötés generálása során helyek egy $P' \subseteq P$ halmazára összesen n token szükséges. Ekkor a következő egyenlőtlenségeket írhatjuk fel:

$$\begin{array}{rcl} \sum_{p_i \in P'} m'(p_i) & \geq & n \\ m'(p_j) & \geq & 0 \; (p_j \in P) \end{array} \tag{3.13}$$

Az első egyenlőtlenség garantálja, hogy P' helyein összesen meglegyen a szükséges tokenszám, miközben a többi azt biztosítja, hogy egyik helyen se legyen negatív számú token. A kapott egyenlőtlenségek együtthatóit a egy A mátrixba és b vektorba gyűjtve a problémát már odaadhatjuk az ILP eszköznek.

Amennyiben nincs megoldás, akkor nem érhetünk el a kiinduló állapotból olyan állapotot, ahol P_i helyein legalább n token van. Ekkor a további megkötések keresése felesleges ehhez a részleges megoldáshoz.

3.4. Az algoritmus egészében

A 4. algoritmus a korábban bemutatott rész-algoritmusokat mutatja be együtt.

Algoritmus 4: Teljes algoritmus

bemenet: $m' \in R(PN, m)$ elérhetőségi probléma vagy $SR(PN, m_0, \mathcal{P})$ rész-elérhetőségi probléma

kimenet : A probléma megoldása („elérhető”/„nem elérhető”), vagy „nem eldönthető”

```
1  Q: Feldolgozandó részleges megoldások sora ;
2  P: Minimális megoldásvektorok sora ;
3  S: Összes részleges megoldás halmaza (Tárolás optimalizációhoz) ;
4  P ← állapotegyenlet minimális megoldása ;
5  while P ≠ ∅ do
6      x megoldásvektor kivétele P-ből ;
7      P ← x-ből ugró megkötésekkel közvetlenül elérhető megoldások ;
8      Ugró megkötések átalakítása ;
9      Q ← x-hez tartozó részleges megoldások (Fa felépítése stubborn set
       halmazokkal és állapotalapú részleges rendezéssel) ;
10     while Q ≠ ∅ do
11         PS(C, x, σ, r) részleges megoldás kivétele Q-ból ;
12         if PS ∈ S then continue;
13         else S ← PS;
14         if r = 0 then return „elérhető” ;
15         if PS kiszűrhető T-invariáns alapon then
16             Jobb álapotok keresése, és berakása Q-ba ;
17             continue;
18         end
19         C' = növelő megkötések PS-hez (rész-elérhetőség vizsgálata
           optimalizációval) ;
20         if C' = ∅ then continue;
21         if Állapotegyenlet és C ∪ C' kielégíthető then
22             x' = állapotegyenlet és C ∪ C' minimális megoldása ;
23             P ← x'-ből ugró megkötésekkel közvetlenül elérhető megoldások ;
24             Q ← x'-hez tartozó részleges megoldások (Fa felépítése stubborn set
               halmazokkal és állapotalapú részleges rendezéssel) ;
25         end
26     end
27 end
28 if Kiszűrtünk T-invariáns alapon megoldást then return „nem eldönthető”;
29 else return „nem elérhető”;
```

3.5. Algoritmus teljességének és helyességének vizsgálata

3.5.1. Teljesség vizsgálata

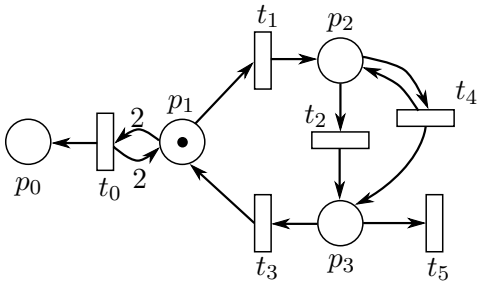
A [17] algoritmus készítői leírták a cikkben, hogy sem a teljességet, sem azt, hogy nem teljes az algoritmus nem tudták bizonyítani. Ezt e-mailben is megerősítették. Ez motivált minket arra, hogy megpróbáljuk bizonyítani, hogy az algoritmus nem teljes. Az eredeti algoritmus teljességére a 3.3. fejezetben több ellenpéldát is bemutattunk. Az algoritmust úgy fejlesztettük tovább, hogy a problémák egy bővebb részhalmazáról tudja eldönteni az elérhetőséget.

Növelő megkötések keresésekor a rész-elérhetőségi probléma vizsgálatával (lásd 3.3.4. fejezet) nem veszítünk el teljes megoldásokat, viszont néhány esetben elkerülhetjük vele a T-invariáns alapú szűrést. Ez azért nagyon előnyös, mert ha nem találunk teljes megoldást, akkor az eredmény „nem elérhető” lesz, tehát el tudjuk dönteni a kérdést, míg ha a T-invariáns alapú szűrés miatt fejeződik be a keresés, akkor nem eldönthető a probléma.

A T-invariáns alapú szűrést a tüzelési sorrend újrarendezésével és a jobb állapotok egy általánosabb definíciójával egészítettük ki. A szűrés így ugyan lassabb lesz, de sokkal több esetben tudja eldönteni az elérhetőséget, ahogy azt az előző fejezet 3.7 és 3.8 ábráin található Petri-hálók is mutatják.

Ellenpélda a teljességre

Az általunk továbbfejlesztett algoritmus sem teljes még, ezt egy konstruktív ellenpéldával be is bizonyítjuk. Tekintsük a 3.10. ábrán látható Petri-hálót és a $(0,1,0,0) \to (1,1,0,0)$ elérhetőségi problémát. Egy kis gondolkodással könnyen megtalálhatjuk a megfelelő $\sigma = (t_1, t_4, t_2, t_3, t_3, t_0, t_1, t_2, t_5)$ tüzelési sorozatot, amellyel elérhetjük a célállapotot. A megoldás lényege, hogy a p_1 helyre közvetetten a t_4 és t_5 tranzíciókkal termelünk és veszünk el egy tokent.



3.10. ábra. Ellenpélda a teljességre.

Amennyiben lefuttatjuk az algoritmust, a minimális megoldásvektor $x_0 = (1,0,0,0,0,0)$, azaz t_0 eltüzelése lesz, melyhez egyetlen $PS_0 = (\emptyset, x_0, \sigma_0 = (), r_0 = (1,0,0,0,0,0))$ részleges megoldás tartozik, ahol t_0-t nem tudjuk eltüzelni. Növelő megkötés keresésekor t_0 miatt p_1 helyre kell egy tokent tenni. Ezt csak t_3 segítségével érhetjük el, így a $|t_3| \geq 1$ megkötéssel az új minimális megoldásvektorba ($x_1 = (1,1,1,1,0,0)$) a t_1, t_2, t_3 invariáns kerül be. Ehhez a $PS_1 = (\{|t_3| \geq 1\}, x_1, \sigma_1 = (t_1, t_2, t_3), r_1 = r_0)$ részleges megoldás tartozik. PS_0 és PS_1 csak egy eltüzelt T-invariánsban különbözik, tehát kiszűrhető. A visszalépegetés során nem találunk jobb állapotot, ugyanis t_0 szempontjából csak a p_1 hely érdekes. Mivel más minimális megoldásvektor vagy részleges megoldás nem volt, az algoritmus nem tudja eldönteni az elérhetőséget.

Fejlesztési lehetőség

Az előbbi háló esetén a probléma oka az, hogy az algoritmus t_0 engedélyezése szempontjából csak p_1 helyet veszi figyelembe és ide szeretne tokent termelni. A p_1 hely azonban p_2-vel és p_3-mal együtt a t_1, t_2, t_3 T-invariáns által érintett helyek közé tartozik, így bármelyikükre termelt token eljuttatható p_1-be.

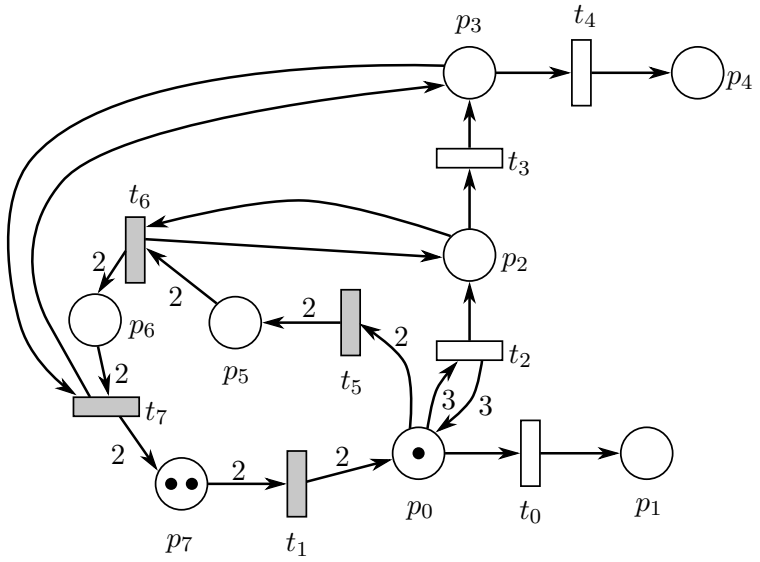
Ezt a gondolatot a következő módon általánosíthatjuk: Tegyük fel, hogy egy részleges megoldást kiszűrtünk T-invariáns alapon. Ekkor nézzük meg, hogy ez a T-invariáns mely $P_T \subseteq P$ helyeken megy át és a visszalépések során számoljuk meg, hogy m_i állapotban maximálisan hány token van egyszerre P_T helyein ($\sum_{p \in P_T} m_i(p)$). Ez után keressük meg azokat a tranzíciókat, amelyek P_T helyei miatt nincsenek engedélyezve és nézzük meg, hogy az előbb kiszámolt maximális tokenszámhoz képest mennyi token hiányzik, illetve ha eltüzelne, mennyi tokent rakna vissza az invariánsra. Ezt az eljárást az összes P_T helyei miatt tüzelni nem tudó tranzícióra megismételve megbecsülhetjük, hogy hány tokent kell P_T helyeire termelni. Ehhez a már ismert módon növelő megkötést generálhatunk és a kapott megoldás irányába egy új ágon elindulhatunk a keresésben.

Az algoritmus teljességének bizonyítása még az előbbi fejlesztési lehetőség után is túlmutatna dolgozatunk keretein.

3.5.2. Helyesség vizsgálata

A 2. tétel kimondja, hogy ha van megoldása az elérhetőségi problémának, az algoritmus meg is találja. A tétel cáfolásaként egy konstruktív ellenpéldát mutatunk amely azt használja ki, hogy a becslésre használt algoritmus néha túlbecsülheti a szükséges tokenszámot.

Tekintsük a 3.11. ábrán látható Petri-hálót az $(1,0,0,0,0,0,0,2) \to (0,1,0,0,1,0,0,2)$ elérhetőségi problémával, azaz szeretnénk elérni, hogy p_0-ból eltűnjön a token és helyette p_1-en és p_4-en megjelenjen egy token. Az ellenpéldát úgy konstruáltuk, hogy a $\sigma_m = (t_1, t_2, t_0, t_5, t_6, t_3, t_7, t_4)$ tüzelési sorozat megoldja az elérhetőségi problémát, tehát egy realizálható megoldásvektor az $x_m = \wp(\sigma_m) = (1,1,1,1,1,1,1,1)$.



3.11. ábra. Ellenpélda a helyességre.

Amennyiben lefuttatjuk az algoritmust, minimális megoldásként az $x = (1,0,1,1,1,0,0,0)$ megoldásvektort találja meg, azaz t_0, t_2, t_3, t_4 eltüzelését. Ezek közül

csak t_0 tüzelhető el, tehát az egyetlen részleges megoldás $PS = (\emptyset, x, \sigma = (t_0), r = (0,0,1,1,1,0,0,0))$ lesz.

Amikor ehhez a részleges megoldáshoz növelő megkötést keresünk, a függőségi gráfban t_2, t_3, t_4 tranzíciók és p_0, p_2, p_3 helyek szerepelnek. Mivel p_2 és p_3 helyekre t_2 illetve t_3 több tokent termel mint vesz el, ezért az egyetlen forrás SCC a p_0-ból álló egy elemű halmaz lesz, ahol jelenleg 0 token van, mivel t_0-t már eltüzeltük. Ekkor az algoritmus azt becsli, hogy 3 token szükséges p_0 helyre. Ide csak t_1 tranzícióval tudunk tokent termelni, amely egy 2 súlyú éllel kapcsolódik p_0-hoz, tehát a kapott megkötés $2|t_1| \geq 3$ lesz. Mivel másfélszer nem tüzelhető egy tranzíció, ezért ez a szürkével jelölt t_1, t_5, t_6, t_7 invariáns kétszeri hozzávételét jelenti.

Ekkor az algoritmus túlbecsülte a szükséges tokenszámot, ugyanis ha az elején t_0-t nem tüzeltük volna el, akkor p_0-ra csak két tokent kellene termelni, ami a szürkével jelölt invariáns egyszeri hozzávételét és az általunk is konstruált x_m megoldásvektor megtalálását jelentené. Ez a túlbecslés azért okoz gondot, mert a szürkével jelölt T-invariánst nem lehet kétszer eltüzelni. A T-invariáns úgy lett megkonstruálva, hogy a t_6 és t_7 tranzíciók csak akkor tüzelhessenek, ha p_2-n illetve p_3-on van token. Először figyeljük meg, hogy ha p_2, p_3, p_4 helyekre token kerül, akkor az onnan nem tud kijutni. Mivel a célállapotban p_2, p_3, p_4 helyeken rendre $0, 0, 1$ tokent szeretnénk látni, ezért 2 token biztosan nem lehet egyszerre ezeken a helyeken. Amennyiben a T-invariánst szeretnénk eltüzelni, előbb egy tokent p_2-re, majd p_3-ra kell rakni. Ez még megoldható egy tokennel, azonban ha újra szeretnénk eltüzelni a T-invariánst, a p_2 helyre egy újabb token kell, ami azt jelentené, hogy már 2 token van összesen p_2, p_3, p_4 helyeken.

Ezzel azt bizonyítottuk, hogy miután az algoritmus a túlbecslés miatt kétszer is hozzávette a szürkével jelölt invariánst, már nem találhat realizálható megoldást.

Megoldás a helyességre

Azt, hogy az algoritmus nem helyes, az okozza, hogy a maximális tüzelési sorozatra való törekvés miatt eltüzelhet olyan tranzíciókat is, amelyeket csak később lenne célszerű (az előbbi példában a t_0 tranzíció). Ez azt fogja okozni, hogy a tüzelési sorozat végállapotában becsült tokenszám nem mindig helyes, amennyiben közben volt több token az adott helyeken.

Továbbfejlesztett algoritmusunkban a tüzelési sorozat közben megvizsgáljuk, hogy az SCC helyein mennyi volt a maximális tokenszám, és ha a végállapotban kevesebb token van, akkor a „nem elérhető” válasz helyett azt mondjuk, hogy túlbecslés lehetősége miatt nem eldönthető a probléma.

4. fejezet

Algoritmus kiterjesztése tiltó éleket tartalmazó Petri-hálókra

Az algoritmust kiterjesztettük, hogy tiltó éles Petri-hálókat is tudjon kezelni. A háttérismeretek között (lásd 2.1.3. fejezet) már foglalkoztunk a tiltó élek kifejezőerejével és azzal is, hogy az elérhetőségi probléma ebben az esetben nem dönthető el. Ebben a fejezetben ismertetjük a tiltó élek kezelése miatti problémákat és megoldásainkat.

4.1. Megoldandó problémák

A tiltó élek kezelése számos problémát felvet. Az algoritmus az állapotegyenletet használja absztrakcióként, ahol a tiltó élek semmilyen formában nem jelennek meg. Az ILP eszköz által szolgáltatott megoldások esetén tehát előfordulhat, hogy tiltó él miatt nem lehet őket realizálni.

Az eredeti algoritmusban amikor egy részleges megoldás nem teljes, akkor növelő megkötések segítségével próbálunk tokent termelni a tüzelni nem tudó tranzíciók engedélyezéséhez. Amennyiben bevezetjük a tiltó éleket, akkor egy tranzíció úgy is lehet nem engedélyezett, hogy valamely tiltó éllel kapcsolódó helyen token van. Ekkor a token termeléssel pont ellentétes módon most tokent kell elvenni helyekről. A token elvételére az egyik lehetőség, hogy az adott helyre tokent termelő tranzíciókra megkötjük, hogy ne tüzeljenek. Ez azonban ellentétben ál az algoritmus eddigi működésével, miszerint egy minimális megoldást fokozatosan bővítve jutunk el a realizálható megoldásig. A másik, általunk is alkalmazott lehetőség az, hogy olyan tranzíciókkal bővítjük a megoldásvektort amelyek az adott helyről tokent vesznek el. Ehhez az eredeti algoritmusból is ismert növelő megkötéseket használjuk, azonban a generálásuk pont ellentétes módon történik, mint a tokent termelő megkötéseké. Ennek a pontos algoritmusát a következő fejezetben ismertetjük.

A tiltó élek kezelésére nem csak az alap algoritmust, hanem az optimalizációkat is fel kell készíteni.

- A stubborn set módszer kiterjesztéséhez nem találtunk megfelelő szakirodalmat, így az jelenleg tiltó élek esetén nem működik.
- Az állapotalapú részleges rendezést nem befolyásolják a tiltó élek, mert ott csak azt vizsgáljuk, hogy ugyanabba az állapotba eljutottunk-e már más tüzelési sorrendben.
- A T-invariáns alapú szűrésnél a visszalépések során a javulás definícióján kell változtatni, mert egy tiltó él miatt nem engedélyezett tranzíció esetén az a jobb, ha az adott helyen kevesebb token van.
- A részleges megoldások tárolását sem befolyásolják a tiltó élek.

- Amikor rész-elérhetőségi problémát vizsgálunk növelő megkötés keresésekor, akkor tiltó élek esetében azt kell vizsgálni, hogy elérhető-e olyan állapot ahol az adott helyen nincs token.

4.2. CEGAR algoritmus kiterjesztése

A megoldandó problémák között már említettük, hogy az alap algoritmust a növelő megkötések keresésénél kell átalakítani. Amikor egy részleges megoldással elakadunk, meg kell vizsgálni, hogy a tüzelni nem tudó tranzíciók miért nem tudnak tüzelni:

- Ha van olyan tranzíció, ami sima él miatt nincs engedélyezve, akkor a 3.2.3. fejezetben ismertetett 3 lépéses algoritmust kell használni a növelő megkötések generálására.
- Ha van olyan tranzíció, ami tiltó él miatt nem tud tüzelni, akkor az előbbi algoritmus egy átalakított változatát kell használni, amelyet a következőkben részletesen ismertetünk.
- Olyan eset is előfordulhat, hogy tiltó él és sima él miatt nem engedélyezett tranzíciók is vannak. Ekkor mindkét algoritmust használni kell, és a kapott megkötések unióját hozzávenni a korábbi megkötésekhez.

4.2.1. Növelő megkötés generálása tiltó élek miatt nem engedélyezett tranzíciókhoz

A 3.2.3. fejezetben ismertetett 3 lépéses növelő megkötés generáló algoritmusnak elkészítettük egy átalakított változatát, ami ellentétes módon, a tokenek elvételét segíti elő adott helyekről.

Tiltó élek esetén is állhat fenn függőség a nem engedélyezett tranzíciók között, ezért az első lépésként az 5. algoritmusban leírt módon egy függőségi gráfot építünk fel.

Algoritmus 5: Függőségi gráf tiltó éles esetre

bemenet: Elérhetőségi probléma $m' \in R(PN_I, m)$; részleges megoldás $ps = (\mathcal{C}, x, \sigma, r)$

kimenet : (P_i, T_i, X_i) struktúrák halmaza, ahol $P_i \subseteq P$, $T_i \cup X_i \subseteq T$

1 $\hat{m}$ kiszámolása: $m[\sigma\rangle\hat{m}$;
2 $G = (P_0 \cup T_0, E)$ páros gráf építése;
3 $T_0 := \{t \in T \mid r(t) > 0 \wedge$ t tiltó él miatt nem engedélyezett$\}$;
4 $P_0 := \{p \in P \mid \exists t \in T_0 : (p,t) \in I \wedge \hat{m}(p) > 0\}$;
5 $E = \{(p,t) \in P_0 \times T_0 \mid (p,t) \in I \wedge \hat{m}(p) > 0\} \cup \{(t,p) \in T_0 \times P_0 \mid W(t,p) < W(p,t)\}$;
6 G erősen összefüggő komponenseinek (SCC) megkeresése;
7 i:=1;
8 **foreach** *forrás SCC (aminek nincs bemenő éle)* **do**
9 $\quad P_i := SCC \cap P_0$;
10 $\quad T_i := SCC \cap T_0$;
11 $\quad X_i := \{t \in T_0 \backslash SCC \mid \exists p \in P_i : (p,t) \in E\}$;
12 $\quad$ i:=i+1;
13 **end**

A gráfba azok a tranzíciók kerülnek, amelyek tiltó él miatt nincsenek engedélyezve. A gráfba felvesszük azokat a helyeket is, amelyek miatt a tranzíciók nem tudnak tüzelni. Egy p helyből t tranzícióba mutató él jelenti, hogy t nem engedélyezett p miatt. A fordított

irányú él azt jelenti, hogy t tüzelése csökkentené p tokenjeinek számát. Most is a forrás SCC-k érdekelnek minket, ugyanis tőlük nem tud egy másik SCC tokeneket elvenni.

A második lépésben megbecsüljük (6. algoritmus), hogy hány tokent kell elvenni az adott SCC helyeiről, hogy a tranzíciók engedélyezetté válhassanak. A becslés eredménye 1 és a tényleges tokenszám között lesz.

Algoritmus 6: Szükséges tokenek számának becslése

bemenet: (P_i, T_i, X_i) struktúra; $m' \in R(PN_I, m)$ elérhetőségi probléma; $\hat{m}$ az előző lépésből

kimenet : Szükséges n tokenszám amit P_i halmaz helyeiről el kell venni

1 **if** $T_i \neq \emptyset$ **then** $n := \min_{t \in T_i}(\sum_{p \in P_i, (p,t) \in I} \hat{m}(p))$;

2 **else** $n := \hat{m}(p)$, ahol $P_i = p$;

Amennyiben T_i halmaz nem üres, megkeressük azt a tranzíciót, amelynek az engedélyezéséhez a legkevesebb tokent kell elvenni. A másik eset ($T_i = \emptyset$) itt egyszerűbb mint sima élek esetén. Mivel P_i egyetlen helyből áll, onnan minden tokent el kell venni, hogy bármelyik tranzíció engedélyezetté válhasson.

Harmadik lépésként a helyekre vonatkozó feltételt át kell alakítani úgy, hogy tranzíciókra vonatkozzon. Legyen $m' \in R(PN_I, m)$ elérhetőségi probléma, $ps = (\mathcal{C}, x, \sigma, r)$ részleges megoldás $r > 0$ maradékvektorral és $\hat{m}$ a σ eltüzelésével kapott állapot $(m[\sigma\rangle\hat{m})$. Legyen P_i a helyek halmaza, ahonnan n tokent kell elvenni, továbbá legyen $T_i := \{t \in T \,|\, r(t) = 0 \wedge \sum_{p \in P_i}(W(p,t) - W(t,p)) > 0\}$. Ekkor a c növelő megkötés a következő alakú:

$$\sum_{t \in T_i} \sum_{p \in P_i} (W(p,t) - W(t,p))|t| \geq n + \sum_{t \in T_i} \sum_{p \in P_i} (W(p,t) - W(t,p))_{\wp}(\sigma)(t) \tag{4.1}$$

A T_i halmazba azok a tranzíciók kerülnek amelyek több tokent vesznek el P_i helyeiről, mint termelnek. Az eredeti algoritmushoz hasonlóan itt is csak olyan tranzíciókat veszünk be T_i-be, amelyek nincsenek benne a maradékvektorban. Az egyenlőtlenség bal oldalán minden $t \in T_i$ tranzíció olyan együtthatóval szerepel, ahány tokent összesen elvesz P_i helyeiről. Jobb oldalt kiszámoljuk, hogy T_i tranzíciói eddig hány tokent vettek el a σ tüzelési sorozat által és ehhez hozzáadjuk a még elvenni kívánt tokenek számát.

A kapott c megkötést a $\mathcal{C}$ halmazhoz adva hasonlóan folytatjuk, mint az alap algoritmus esetében.

4.3. Optimalizációk kiterjesztése

Korábban említettük, hogy az optimalizációk közül a T-invariáns alapú szűrést és a növelő megkötés keresése közbeni rész-elérhetőségi probléma vizsgálatát kell módosítani, hogy együttműködjenek a tiltó élekkel.

T-invariáns alapú szűrés

Amennyiben egy megoldást kiszűrünk és visszalépésekkel jobb állapotokat keresünk, egy tiltó él miatt nem engedélyezett tranzíció szempontjából az számít jobb állapotnak, ha a hozzá tiltó éllel kapcsolódó helyeken kevesebb token van. A javulás definíciója a következőképpen egészül ki: egy m_i állapotot akkor tekintjük jobbnak az m' végállapotnál, ha létezik olyan $t \in T, r(t) > 0$ tranzíció és $p \in {\bullet}t$ hely, amire

$$(m'(p) < W(p,t) \wedge m_i(p) > m'(p)) \quad \vee \quad ((p,t) \in I \wedge m'(p) > 0 \wedge m_i(p) < m'(p)) \tag{4.2}$$

teljesül. A plusz feltétel azt fejezi ki, hogy t tranzíció a (p, t) tiltó él miatt nem tud tüzelni és a köztes állapotban kevesebb token van a p helyen, mint a végállapotban.

Rész-elérhetőségi probléma vizsgálata növelő megkötés keresésekor

Amennyiben tiltó élek miatt nem engedélyezett tranzíciókhoz keresünk növelő megkötést, az eredeti algoritmushoz hasonlóan leellenőrizhetjük, hogy az állapotegyenlet alapján el tudunk-e adott mennyiségű tokent venni a kérdéses helyekről. Amennyiben helyek egy P' halmazán legfeljebb n token lehet, formálisan az alábbi egyenlőtlenségeket írhatjuk fel:

$$\begin{array}{rcl} \sum_{p_i \in P'} m'(p_i) & \leq & n \\ m'(p_j) & \geq & 0 \, (p_j \in P) \end{array} \tag{4.3}$$

A felső egyenlőtlenség biztosítja, hogy P' helyein összesen legfeljebb n token legyen. Az alsó egyenlőtlenségre azért van szükség, hogy egyik helyen se lehessen negatív mennyiségű token.

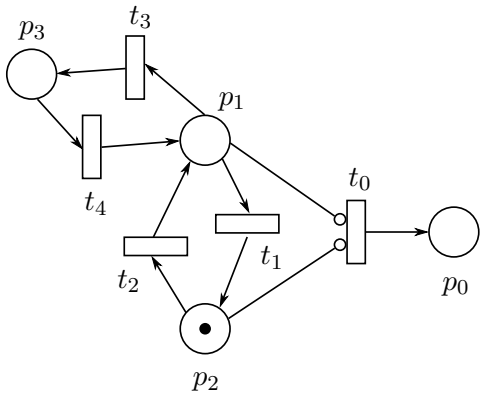
Amennyiben ennek a rész-elérhetőségi problémának nincs megoldása, a kiinduló állapotból nem érhetünk el olyan állapotot, ahol P' helyein összesen legfeljebb n token van, tehát fölösleges is megkötéseket keresni.

Mivel a tiltó élek nem jelennek meg az állapotegyenletben, ez az optimalizáció kevesebbszer fog eredményesen szűrni.

4.3.1. Példa háló

A fejezet végén egy példával is bemutatjuk a kiterjesztett algoritmus működését. A példában T-invariáns alapú szűrésre is sor kerül.

10. példa. *Tekintsük a 4.1. ábrán látható tiltó éles Petri-hálót és a* $(0, 0, 1, 0) \to (1, 0, 1, 0)$ *elérhetőségi problémát.*



4.1. ábra. Példa tiltó éles hálóra.

A minimális megoldás $x_0 = (1, 0, 0, 0, 0)$ *azaz* t_0 *eltüzelése. Mivel* t_0 *nem engedélyezett a* (p_2, t_0) *tiltó él miatt, egyetlen* $PS_0 = (\emptyset, x_0, \sigma_0 = (), r_0 = (1, 0, 0, 0, 0))$ *részleges megoldást találunk. Növelő megkötés keresésekor az algoritmus megtalálja, hogy* t_0 *a* p_2*-n lévő token miatt nem tud tüzelni, ezért megpróbálja azt onnan elvenni. A* p_2 *helyről egyedül* t_2 *tranzíció vesz el tokent, ezért a kapott megkötés* $|t_2| \geq 1$ *alakú. Az új* $x_1 = (1, 1, 1, 0, 0)$ *megoldásvektorba bekerül a* t_1, t_2 *invariáns. Ehhez egyetlen* $PS_1 = (\{|t_2| \geq 1\}, x_1, \sigma_1 = (t_2, t_1), r_1 = r_0)$ *részleges megoldás tartozik, mivel a kezdőállapotban a megoldásvektorban is szereplő tranzíciók közül csak* t_2 *engedélyezett, a tüzelése után pedig csak* t_1*, mivel* (p_1, t_0) *tiltó él megakadályozza* t_0 *engedélyezését.*

Amennyiben nem alkalmaznánk a T-invariáns alapú szűrést, az algoritmus itt végtelen ciklusba kerülne. Ezen optimalizáció alkalmazásával azonban elkerülhetjük a végtelen ciklust, sőt a köztes állapotok között megtaláljuk a t_2 tüzelése után szereplő $(0, 1, 0, 0)$ állapotot, amely t_0 szempontjából jobb, mint a végállapot, hiszen a p_2 helyen kevesebb token van.

Az ehhez tartozó $PS_2 = (\{|t_2| \geq 1\}, x_2 = x_1, \sigma_2 = (t_2), r_2 = (1, 1, 0, 0, 0))$ részleges megoldásból folytatva az algoritmus rájön, hogy t_0 engedélyezéséhez p_1-ből el kell venni a tokent. Erről a helyről t_3 és t_1 is el tudja venni a tokent, de mivel utóbbi benne van a maradékvektorban, ezért a kapott megkötés $|t_3| \geq 1$ alakú. Amennyiben ezzel a megkötéssel kiegészítve megoldjuk az állapotegyenletet, a kapott $x_3 = (1, 1, 1, 1, 1)$ megoldás a $\sigma_3 = (t_2, t_3, t_0, t_4, t_1)$ tüzelési sorozattal realizálható.

5. fejezet

Megvalósítás

Ebben a fejezetben nagy vonalakban bemutatjuk az algoritmus implementációját (lásd 5.2. fejezet) és a használt keretrendszert (lásd 5.1. fejezet).

5.1. Keretrendszer

Az algoritmus megvalósításához a PetriDotNet keretrendszert [2] és az Lpsolve ILP megoldót használtuk, melyeket ebben a fejezetben ismertetünk.

5.1.1. PetriDotNet

A PetriDotNet keretrendszer Petri-hálók szerkesztésére, szimulációjára és analízisére szolgál. A program C# nyelven íródott és publikus interfészek segítségével kiegészítőkkel bővíthető. Az algoritmust is egy kiegészítőként implementáltuk, amely a PetriDotNet-ben szerkesztett hálóra képes vizsgálni az elérhetőségi problémát.

Publikus interfész

A PetriDotNet kiegészítőknek tartalmazniuk kell egy osztályt amely megvalósítja a *IPDNPlugin* interfészt. Ez az osztály kap egy *PDNAppDescriptor* típusú referenciát az aktuális alkalmazásra, amely *CurrentPetriNet* attribútumából érhetjük el az aktuális Petri-hálót. Az algoritmus a *PetriNet* típusú *CurrentPetriNet* attribútum alapján saját adatszerkezeteket épít fel a Petri-háló tárolására és utána csak ezekkel dolgozik. Az adatszerkezetek felépítéséhez a *PetriNet* osztály alábbi attribútumait és metódusait használja:

- A *GetPlaces()* metódus egy *Place* listában adja vissza a háló helyeit. Az algoritmus a helyeket saját sorszámmal látja el, de a helyek nevét (*Name* attribútum) is megjegyzi a grafikus felületen történő azonosítás miatt.
- A *GetTransitions()* metódus a *GetPlaces()*-el analóg módon egy *Transition* listában adja vissza a hálóban szereplő tranzíciókat, melyet az algoritmus a helyekhez hasonlóan egy sorszámmal kezel.
- A szomszédossági mátrixhoz az élekre is szükség van, amelyeket a *GetNonVisualEdges()* segítségével egy *Edge* listában kapunk meg. Az élek rendelkeznek *Source* és *Target* attribútumokkal, melyek a forrás illetve cél helyet/tranzíciót adják meg. Az él súlyát a *Weight* egész szám tartalmazza.

Az algoritmus ezek után a saját adatszerkezeteivel dolgozik, melyeket a következő fejezetben (5.2.) ismertetünk.

5.1.2. Lpsolve

Az algoritmus futása során az ILP problémákat a nyílt forráskódú Lpsolve eszköz 5.5 verziójával oldjuk meg. Az Lpsolve rendelkezik C# interfésszel, amely a honlapról [1] letölthető. A honlapon ezen kívül egy részletes dokumentáció is található az eszközről. Dolgozatunkban csak az általunk használt funkciókat ismertetjük.

Az Lpsolve összes funkcionalitását az *lpsolve55* névtér statikus *lpsolve* osztálya biztosítja. Először az *init* függvény segítségével inicializálni kell az Lpsolve-t. A függvény paramétere az *lpsolve55.dll* fájl elérési útvonala. Az inicializálás után a *make_lp* függvénnyel hozhatunk létre új LP problémát. Lpsolve-ban egy LP problémát megkötésekkel írhatunk le, amelyek a változókra vonatkozó lineáris egyenletek vagy egyenlőtlenségek. A megkötéseket soroknak, a változókat oszlopoknak nevezik. A *make_lp* első paramétere a sorok száma (kezdetben 0), a második a változók (esetünkben tranzíciók) száma, a visszatérési érték pedig egy *int* típusú változó amivel ezek után az imént létrehozott problémára hivatkozhatunk. Ha a létrehozás során hiba történt, akkor a visszatérési érték 0.

Miután megvan az LP problémánk, hozzá kell adni a sorokat. Erre az *add_constraint* függvény szolgál, amely a következő 4 paraméterrel rendelkezik:

- *lp*: Az LP probléma azonosítója amelyhez a megkötést adjuk.
- *row*: A változók együtthatóinak tömbje.
- *constr_type*: A megkötés típusa. Lehet egyenlőség (EQ), kisebb-egyenlő (LE) vagy nagyobb-egyenlő (GE)[1].
- *rh*: A megkötés jobb oldalán álló konstans.

A megkötések után az optimalizáció célfüggvényét a *set_obj_fn* függvénnyel állíthatjuk be. Első paraméterként az LP probléma azonosítóját kell megadni, másodikként pedig a változók együtthatóját a célfüggvényben. A *set_minim* függvénnyel megmondhatjuk, hogy minimalizálásra törekszünk.

Mielőtt megoldanánk a problémát, be kell állítani, hogy a változóink egészértékűek legyenek. Ezt a *set_int* függvénnyel tehetjük meg, melynek az LP probléma azonosítóját, a változó sorszámát és egy „igaz” logikai értéket kell megadni minden változó esetén.

A problémát a *solve* függvénnyel oldhatjuk meg, melynek egyetlen paramétere a probléma azonosítója. A függvény visszatérési értékben jelzi a megoldás sikerességét. Az eszköz honlapján az összes lehetséges visszatérési értékét leírják, de mi csak az *OPTIMAL* és *INFEASIBLE* esetekkel foglalkozunk, a többi esetben hibát jelzünk. *INFEASIBLE* visszatérési érték esetén tudjuk, hogy nem létezik megoldása az ILP problémának. *OPTIMAL* visszatérés után a *get_variables* függvénnyel kérdezhetjük le a megoldást. Első paraméter a probléma azonosítója, második pedig a tömb, amibe a megoldást szeretnénk megkapni.

A probléma megoldása után egy *delete_lp* hívással szabadíthatjuk fel az erőforrásokat.

5.2. CEGAR algoritmus

Ebben a fejezetben bemutatjuk az általunk fejlesztett szoftver szerkezetét.

[1]Bár az ILP problémánál formálisan nagyobb-egyenlő a megkötés típusa, de a kisebb-egyenlő és az egyenlőség is átalakítható nagyobb-egyenlő típusúra. Az Lpsolve ezt helyettünk elvégzi, így az implementáció során a kényelmesség miatt megengedtük mindhárom típusú megkötés használatát.

5.2.1. Osztályok és interfészek

Ebben a fejezetben felsoroljuk az implementáció során használt osztályokat, interfészeket és azok főbb funkcióit, felelősségeit. Az átláthatóság érdekében az osztályokat és interfészeket feladatuk alapján több csoportba soroljuk.

Fő osztályok

- **Plugin** - A PetriDotNet keretrendszer által elvárt interfészt valósítja meg. Összeköti az általunk készített programot a keretrendszerrel. A PetriDotNet megfelelő menüpontjának kiválasztásakor elindítja a programot és átadja az éppen szerkesztés alatt álló Petri-hálót.

- **Reachability** - Ez a fő osztály, amely az elérhetőségi analízist végzi. Konstruktorában megkapja az elérhetőségi probléma paramétereit (Petri-háló, kiinduló állapot, cél állapot vagy predikátumok), melyeket eltárol. *IsReachable* függvényének meghívásával indítjuk el az analízist, melynek során a többi segédosztályt használja a feladat megoldására.

- **Config** - Statikus osztály, amely a beállításokat tárolja. Segítségével szabályozhatjuk az optimalizációkat és a naplózás részletességét.

Tároláshoz használt adatszerkezetek

- **PetriNetMatrix** - Petri-hálót reprezentáló osztály. Mátrixok segítségével tárolja a Petri-háló adatait (élsúlyok, szomszédossági mátrix, tiltó élek). A belső reprezentációban a helyeket és a tranzíciókat 0-tól indexelve tárolja. Számos lekérdezőfüggvényt biztosít az adatok elérésére, illetve képes adott tokeneloszlás mellett eldönteni, hogy mely tranzíciók tüzelhetnek és mi lenne a tüzelés eredménye. A stubborn set optimalizáció is itt van megvalósítva, lekérdezhetjük adott tokeneloszlás mellett a stubborn set-ben lévő tranzíciók halmazát.

- **Vector** - Tetszőleges hosszú vektor tárolására képes osztály. Az értékek tárolása mellett számos segédfüggvénnyel rendelkezik, például komparálással és hash érték számolással.

- **Predicate** - Helyekre vonatkozó predikátum ($Am \geq b$) egy sorát reprezentálja. Tárolja a sor együtthatóit, az operátorát ($\geq, =, \leq$) és a jobboldali értékét.

- **MarkingRemainderPair** - Tokeneloszlás és maradékvektor párok tárolására szolgáló egyszerű segédosztály.

- **SolutionConstrListPair** - Megoldásvektor és megkötéslista párok tárolására szolgáló egyszerű segédosztály.

Megkötések

- **IConstraint** - Megkötések közös interfésze. Biztosítja az együtthatók, az operátor és a jobboldali érték elérését olyan formában, ahogy az Lpsolve várja.

- **IncrementConstraint** - $\sum_{i=1}^{k} n_i|t_i| \geq n$ alakú növelő megkötést reprezentáló osztály, amely megvalósítja az *IConstraint* interfészt. Tárolja a tranzíciók együtthatóit és a jobboldali értéket. Az operátor mindig „$\geq$” típusú.

- **JumpConstraint** - $|t_i| < n$ alakú ugró megkötést reprezentáló osztály, amely megvalósítja az *IConstraint* interfészt. Tárolja, hogy melyik tranzícióra vonatkozik és a jobboldali értéket. Az operátor mindig „$\leq$” típusú.

- **PredicateConstraint** - Helyekre vonatkozó predikátum átalakított formáját reprezentálja, ami már tranzíciókra vonatkozik. Tárolja az együtthatókat, az operátort és a jobboldali értéket.

- **ConstrList** - Megkötések listáját tároló osztály. Képes a listában szereplő ugró megkötéseket növelővé alakítani és a tárolás optimalizációban említett módon a fölösleges megkötéseket eltávolítani.

- **IncrementBuilder** - Növelő megkötések létrehozását segítő osztály. Megvalósítja a 3.2.3. fejezetben ismertetett 3 lépésből álló növelő megkötés generáló algoritmust. Az SCC-k kereséséhez a *Graph* osztályt használja.

SCC keresés adatszerkezetei

- **Graph** - Az SCC keresésre szolgáló gráfot reprezentáló osztály. Képes megkeresni a gráf erősen összefüggő komponenseit és a hozzájuk tartozó (P_i, T_i, X_i) struktúrákat.

- **GraphNode** - Az SCC keresésre szolgáló gráf egy csúcsa. Tárolja az azonosítóját, a szomszédait és néhány járulékos adatot az SCC kereséshez.

- **STXtuple** - (P_i, T_i, X_i) struktúrát reprezentáló osztály.

Részleges megoldások adatszerkezetei

- **PartialSolution** - $(\mathcal{C}, x, \sigma, r)$ alakú részleges megoldást reprezentáló osztály. A megkötéslista, megoldásvektor, tüzelési sorozat és maradékvektor mellett számos olyan adatot tárol amely az algoritmust és az optimalizációkat segíti. Ilyenek például a tüzelési sorozat végén elért tokeneloszlás és az, hogy teljes-e a megoldás.

- **MarkingTreeNode** - A részleges megoldások generálása során bejárt fa egy csomópontja (lásd 3.2.4. fejezet). Tárolja az elért tokeneloszlást, maradékvektort és néhány egyéb segédadatot az algoritmushoz.

- **PartialSolutionCatalog** - Részleges megoldások tárolását megvalósító osztály. Az optimalizációk (lásd 3.2.5. fejezet) között leírt módon sorolja ekvivalenciaosztályokba a részleges megoldásokat.

- **PartialSolutionTrackerNode** - A T-invariáns alapú szűrés optimalizáció segéd adatszerkezete. Egy fában tartja nyilván az elért részleges megoldásokat. Képes eldönteni egy részleges megoldásról, hogy kiszűrhető-e és ha igen, akkor vannak-e olyan köztes állapotok, ahonnan érdemes folytatni.

LPsolve segédosztályai

- **lpsolve55** - Az Lpsolve interfészét biztosító osztály.

- **LpSolveException** - Saját kivételosztály az Lpsolve hibáinak esetleges kezelésére.

- **LpSolveTool** - Statikus segédosztály az Lpsolve használatához. Egyik függvénye (*Solve*) a kapott állapotegyenlet (vagy predikátumok) és megkötések alapján felépíti

az ILP problémát és visszaadja a megoldást, ha van. A másik függvény (*StateEquationTest*) a növelő megkötés keresése közbeni rész-elérhetőségi probléma teljesülését vizsgálja (lásd 3.3.4. fejezet).

Grafikus felület

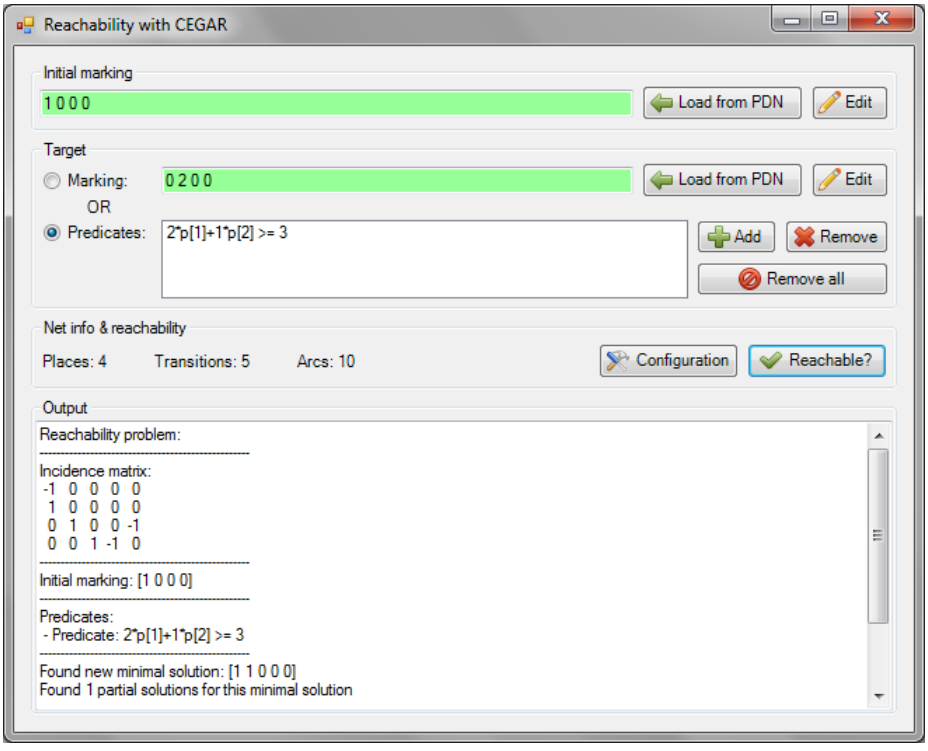
- **PluginForm** - A program főablaka (lásd 5.1. ábra). A kezdő tokeneloszlást az *Initial marking* szövegdobozba kell beírni vektor formájában. A *Load from PDN* gomb hatására a grafikus felület aktuális tokeneloszlását tölti be, az *Edit* gomb segítségével pedig egy táblázatos szerkesztőben adhatók meg az egyes helyek tokenszámai. Elérhetőségi probléma esetén a *Target* csoport *Marking* szövegdobozát hasonlóan lehet kitölteni. Rész-elérhetőségi probléma vizsgálatakor a *Target* csoport *Predicates* listájában láthatóak a predikátumok. Az *Add* és *Remove* gombokkal lehet hozzávenni illetve törölni. A *Net info & reachability* csoportban megjelennek a háló adatai. A *Configuration* gombra kattintva előjön a beállítások ablaka. Az elérhetőségi analízis a *Reachable?* gombra kattintva indítható. A program beállításoktól függően az *Output* csoport szövegdobozába naplóz.

- **AddPredicate** - Predikátum kényelmes szerkesztésére szolgáló ablak. Grafikus felületen állíthatók be az együtthatók, az operátor és a jobboldali érték.

- **ConfigForm** - Beállítások ablaka.

- **MarkingEditor** - Tokeneloszlás kényelmes szerkesztésére szolgáló ablak.

- **ConsoleHandler** - Segédosztály a naplózás egységes kezeléséhez. Globálisan mindig elérhető pontosan egy példányban (singleton minta).



5.1. ábra. A program főablaka

Az osztályok között leginkább függőség van, melyekre az adott osztály ismertetésénél kitértünk. Az osztálydiagram nem fejezné ki elég jól a program szerkezetét, ezért azt nem készítettünk.

6. fejezet

Mérési eredmények

Ebben a fejezetben az algoritmus hatékonyságát vizsgáljuk különböző problémákra. Összehasonlító méréseket végeztünk más eszközökkel, algoritmusokkal, és a mi algoritmusunk futását is megvizsgáltuk különböző beállításokkal. A mérések során teljes és rész elérhetőségi problémákat is vizsgálunk, emellett bemutatjuk az új, tiltó éleket is tartalmazó Petri-háló elérhetőségi algoritmus futási eredményeit is. A mérési eredményeket ismertető táblázatainkban *PDN* cimkével jelöljük a PetriDotNet környezetben mért saját eredményeinket. A 6.1. fejezetben bemutatjuk az algoritmus működését teljes elérhetőségi problémákra, illetve összevetjük az eredeti algoritmust implementáló *Sara* eszközzel is. A 6.2. fejezetben bemutatjuk az algoritmus skálázódását, a 6.3. fejezetben pedig érzékeltetjük az egyes optimalizációk hatását. A 6.4. fejezetben az általunk kifejlesztett új megközelítést összehasonlítjuk más algoritmusokkal, végül a 6.5. fejezetben megvizsgáljuk, hogy egyszerűbb tiltó éleket tartalmazó Petri-hálókra milyen sebességgel fut le az új algoritmus.

A saját algoritmusunk méréseit a következő konfiguráción végeztük el: virtuális gépben futó Windows 8 (x32), 2.2 GHz processzor (Intel-T4400), 1GB memória, .Net 4.0 futtatókörnyezet. A mérések során az algoritmus memóriafogyasztása nem haladta meg 600 MByte-ot, ezért ez a konfiguráció is megfelelő volt. A Sara eszközhöz is hasonló konfigurációt használtunk Debian (x32) operációs rendszerrel.

Az eredményeket összesítő táblázatokban az egyes mezők „futási idő/megjegyzés” formátumúak, ahol a megjegyzés lehet:

- **nincs**: az algoritmus jól lefutott
- **CD**: nem tudott dönteni
- **NR**: a futási idő $>$ 600 másodperc

A mérések során használt modellek mögöttes tartalma és hogy milyen forrásokból származnak:

- **ConsumerProducer** (CP): Két párhuzamos termelést modellez. Forrás: mi állítottuk össze.
- **FMS**: Ez a rendszer egy rugalmas gyártórendszert modellez [4].
- **MAPK**: Ez a Petri-háló egy biokémiai reakciót modellez [3].
- **Kanban**: Ez a háló egy termelés vezérlési módszert mutat be [4].
- **Étkező Filozófusok** (DPhil): Szinkronizációs problémák szemléltetésére használt modell [4].
- **SlottedRing** (SR): Egy hálózati protokoll működését modellezi [8].
- **Hanoi Tornyai**: Egy ismert matematikai játék [8].

6.1. Összehasonlítás más CEGAR megközelítéssel

Az algoritmust egy másik CEGAR megközelítést implementáló eszközzel, a Sara eszközzel [3] vetjük össze. Ez az eredeti algoritmust implementálja [17]. A 3.3. fejezetben már bemutattuk, hogy bizonyos hálókra ez az algoritmus az elérhetőségi problémát nem tudja megoldani, a mi algoritmusunk pedig igen, illetve, hogy a Sara eszközben implementált algoritmus a 3.3.3. fejezetben szereplő hálóra hibás eredményt ad. A fejlesztéseink során kijavítottuk ezt az algoritmikus hiányosságot (lásd 3.3. fejezet). Ebben a fejezetben elsősorban a helyes működés esetén tapasztalt teljesítményt vizsgáljuk, ezért olyan Petri-háló modelleket választottunk, amelyek esetén az eredeti algoritmus is helyesen fut.
A mérések eredménye a következő táblázatban látható.

Háló neve	**Sara**	**PDN**
CP_500	0,2s	0,5s
CP_NR_10	0,2s	0,5s
CP_NR_25	111s	2s
CP_NR_50	NR	16s
Kanban_1000	0,2s	1s
FMS_1500	0,5s	5s
MAPK	0,2s	1s

Az első modellek mérési adataiból azt a következtetést szűrhetjük le, hogy az általunk megvalósított részleges rendezéses algoritmus hatékonyabb, mint a Sara eszközben implementált változat. A ConsumerProducer modell konkurens, ezért a mi algoritmusunk jobban skálázódik. A Kanban, FMS és MAPK modellek esetén látható teljesítmény hátrány oka az, hogy mi a fejlesztéseinket magasabb szintű programozási nyelven végeztünk, míg a Sara eszköz C programozási nyelven készült.

6.2. Az algoritmus skálázódása

A következő mérések során az algoritmus skálázódását vizsgáljuk néhány ismert Petri-hálóra. A mérési eredményeinket a következő táblázat tartalmazza.

Háló neve	**Futási idő**
CP_50	0,2s
CP_500	0,5s
CP_NR_18	1s
CP_NR_25	2s
CP_NR_50	16s
Kanban_100	0,4s
Kanban_1000	1s
SR_10	1s
SR_20	42s
SR_50	433s
SR_100	6772s
DPhil_10	0,2s
DPhil_20	1s
DPhil_50	45s
DPhil_100	535s
FMS_100	0,5s
FMS_1500	5s

Az eredmények értékelése: Mivel az ILP megoldás megtalálása NP teljes probléma, ezért sok helyből és tranzícióból álló hálókra az algoritmus láthatóan lassul (Étkező Filozófus és a SlottedRing modellek). A kis méretű, de nagy állapottérrel rendelkező hálók esetén, ahol más módszerek az állapotér mérete miatt küzdenek gondokkal, nagyon jól teljesít a CEGAR alapú megközelítés: a Kanban és FMS modelleknél lineárisan skálázódik.

6.3. Optimalizációk hatása

A következő táblázatban az optimalizációk hatását kívánjuk szemléltetni. Az egyes oszlopok jelentése:

- **Teljes**: összes optimalizációval,
- **¬O1**: stubborn set halmazok használata nélkül,
- **¬O2**: állapotalapú részleges rendezés nélkül,
- **¬O3**: T-invariáns alapú szűrés nélkül,
- **¬O4**: részleges megoldások tárolása nélkül,
- **¬OA**: összes optimalizáció nélkül.

Háló neve	**Teljes**	**¬O1**	**¬O2**	**¬O3**	**¬O4**	**¬OA**
CR_NR_8	0,1s	26s	1s	0,1s	0,1s	NR
CR_NR_10	0,5s	20s	93s	0,5s	0,5s	NR
ConsumerProducer2NR_25	2s	NR	NR	2s	2s	NR
Kanban_1000	1s	1s	1s	1s	1s	1s
SlottedRing_50	433s	435s	437s	NR	435s	NR
DPhil_50	45s	45s	45s	45s	45s	45s

Ez első három mérésből látszik, hogy a stubborn set halmazok és az állapotalapú részleges rendezés alkalmazása is nagy mértékben csökkentheti a futási időt. A SlottedRing_50 hálónál a T-invariáns alapú szűrés segít nagyon sokat. (Nélküle végtelen ciklusba kerül az algoritmus.) A Kanban és Étkező Filozófusok problémáknál pedig látható, hogy az egyes optimalizációknak nincs nagy számításigénye az alap algoritmushoz képest.

6.4. Összehasonlítás más algoritmusokkal

Az általunk fejlesztett algoritmust összehasonlítottuk egy másik, Petri-hálókra nagyon hatékony szimbolikus állapottér generáló algoritmussal [8], az úgynevezett szaturációs algoritmussal. Ez az algoritmus a hatékonyságát egy speciális, a Petri-hálók sajátosságaihoz kiválóan illeszkedő speciális iterációs stratégiának köszönheti.

Háló neve	**Szaturáció**	**PDN**
Kanban_1000	NR	1s
SlottedRing_50	4s	433s
DPhil_50	0,5s	45s
FMS_1500	NR	5s

Jól látható a mérésekből, hogy azokban az esetekben, amikor az ILP megoldó nem hatékony, akkor a szimbolikus módszerek jól teljesítenek: ilyenek a nagy, aszinkron hálók. Ilyenkor általában a keresett tulajdonság megtalálásához nem kell mély állapotteret bejárni, ellenben az ILP probléma megoldása a sok változónak köszönhetően költséges. Azokban az esetekben, amikor az állapottér bonyolult, és az elérhetőségi probléma megoldásához hosszú trajektóriát kell bejárni, egyértelműen hatékonyabb a CEGAR megközelítés. Emellett fontos megjegyezni, hogy ezek a modellek végesek, hiszen a szaturációs algoritmus véges állapotterű modelleket tud hatékonyan kezelni, míg a mi megközelítésünk akár végtelen állapotterű modellekkel is megbirkózik.

6.5. Tiltó éleket tartalmazó Petri-hálók

Megvizsgáltuk az új algoritmusunkat tiltó éleket tartalmazó Petri-hálóra. Ezek között voltak, amiket egyszerű hálókból transzformáltunk, így összehasonlíthatjuk az új algoritmus teljesítményét az egyszerű Petri-hálókat kezelő CEGAR algoritmussal. Mérési eredményeinket az alábbi táblázat tartalmazza.

Háló neve	**Futási idő**
Hanoi_2	1s
Test1	0,5s
Test2	27s

A Hanoi tornyai néven ismert matematika feladványt modellező háló esetén egy gondolkodtató problémát oldottunk meg az algoritmusunkkal. A Test1 (lásd 4.1. ábra) és Test2 (a Test1 példa bővítése t_0-hoz több tiltó éllel kapcsolódó hely hozzávételével, amelyek jelentősen bonyolítják az elérhetőségi analízist) hálók a T-invariáns alapú szűrés optimalizációt aktívan használó, általunk összerakott hálók voltak. Az optimalizáció tiltó élek esetén is olyan teljesítménnyel futott, mint az ezekhez a hálókhoz hasonló, nem tiltó éles modellek esetén. A tiltó éleket is tartalmazó Petri-hálók elérhetőségi problémája általános esetben nem dönthető el, azonban az algoritmus további fejlesztésével szeretnénk elérni, hogy nagyobb modellekre is lefusson.

7. fejezet

Összefoglalás

A Petri-hálók a matematikai modellezés elterjedt eszközei. Gyakran használják rendszerek modellezésére és analízisére [14], emellett sok eldönthetőségi probléma is visszavezethető a Petri-hálók elérhetőségi problémájára [5].

Munkánk célja egy hatékony algoritmus kifejlesztése volt a Petri-háló elérhetőség eldöntésére. Ez egy EXSPACE-nehéz probléma, tehát nem adható olyan algoritmus, amely minden esetre hatékonyan lefutna. Ezért komoly algoritmikus kihívás ennek megoldása. Munkánk kiindulási pontja egy új megközelítés, amely az eddigi algoritmusokhoz képest egy ezen a területen új ötletet, a matematikai absztrakció eszközét (CEGAR) használja fel [17]. Ez az algoritmus több kategóriában is megnyerte a 2011. évi modellellenőrző versenyt [4]. A CEGAR jellegű megközelítések előnye a hatékonyság, azon az áron, hogy általában nem szokták garantálni az algoritmikus teljességet. Ilyen jellegű vizsgálatot azonban a munkánk alapját képező algoritmuson senki sem végzett előttünk. Dolgozatunkban megvizsgáltuk ezt az új megközelítést, és elkészítettünk egy saját implementációt, amiben az algoritmus különböző hiányosságait javítottuk és az algoritmust továbbfejlesztettük. Eredményeink három kategóriába sorolhatók: elméleti, alkalmazhatóság-bővítési és gyakorlati jellegű eredményeket értünk el. A dolgozatunkban bemutatott elméleti eredményeink az alábbiak:

- bizonyítottuk, hogy az algoritmus nem teljes,
- bizonyítottuk, hogy az algoritmus egyik fő heurisztikája bizonyos kisarkított helyzetekben nem helyes,
- algoritmikus fejlesztéseinkkel lehetővé tettük egyszerű Petri-hálók szélesebb körének elérhetőségi analízisét,
- algoritmikus fejlesztéseinkkel garantáljuk a helyes eredményt,
- az optimalizációk területén végrehajtott fejlesztésünkkel gyorsítottuk az algoritmus futását.

A meglévő algoritmust adaptáltuk új típusú problémaosztályok kezelésére:

- lehetővé tettük Petri-hálók predikátumokkal definiált rész-elérhetőségi problémáinak vizsgálatát,
- kiterjesztettük az algoritmust tiltó élekkel rendelkező Petri-hálók kezelésére.

A dolgozatban bemutatott gyakorlati eredményeink:

- elkészítettük az algoritmusok implementációját a tanszéken fejlesztett keretrendszerben,
- mérésekkel megvizsgáltuk az algoritmus hatékonyságát, melyet összevetettünk más eszközökkel is.

A jövőben szeretnénk elérni egyszerű Petri-hálók estén az algoritmikus teljességet, amely azért nagy kihívás, mert tudomásunk szerint jelenleg nem létezik olyan eszköz, amely ezt garantálná. Emellett a mérési eredmények azt sugallják, hogy sok esetben a hatékonyságon is van lehetőség javítani. A tiltó éleket is tartalmazó Petri-hálók elérhetőségi analízisét vizsgáló algoritmuson is szeretnénk tovább gyorsítani, hogy nagyobb modellek vizsgálatára is alkalmas legyen.

Ábrák jegyzéke

Irodalomjegyzék

[1] lpsolve honlapja (utoljára megtekintve: 2012.10.26.)
`http://sourceforge.net/projects/lpsolve/`.

[2] A petridotnet keretrendszer honlapja. (utoljára megtekintve: 2012.10.26.)
`http://petridotnet.inf.mit.bme.hu/`.

[3] Sara eszköz honlapja (utoljára megtekintve: 2012.10.26.)
`http://service-technology.org/tools/download/`.

[4] Sumo'2011 model checking contest (utoljára megtekintve: 2012.10.26.)
`http://sumo.lip6.fr/Model_Checking_Contest.html`.

[5] Mohamed Faouzi Atig, Ahmed Bouajjani, and Shaz Qadeer. Context-bounded analysis for concurrent programs with dynamic creation of threads. TACAS '09, pages 107–123, Berlin, Heidelberg, 2009. Springer-Verlag.

[6] Piotr Chrzšstowski-Wachtel. Testing undecidability of the reachability in petri nets with the help of 10th hilbert problem. In Susanna Donatelli and Jetty Kleijn, editors, *Application and Theory of Petri Nets 1999*, volume 1639 of *Lecture Notes in Computer Science*, pages 690–690. Springer Berlin / Heidelberg, 1999.

[7] George B. Dantzig and Mukund N. Thapa. *Linear programming 1: introduction.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1997.

[8] Darvas Dániel and Jámbor Attila. Komplex rendszerek modellezése és verifikációja. 2011.

[9] Javier Esparza, Stephan Melzer, and Joseph Sifakis. Verification of safety properties using integer programming: Beyond the state equation, 1997.

[10] K. Schmidt L. M. Kristensen and A. Valmari. Question-guided stubborn set methods for state properties, 2006.

[11] R.J. Lipton. *The Reachability Problem Requires Exponential Space.* Research report (Yale University. Dept. of Computer Science). Department of Computer Science, Yale University, 1976.

[12] Ernst W. Mayr. An algorithm for the general petri net reachability problem. In *Proceedings of the thirteenth annual ACM symposium on Theory of computing*, STOC '81, pages 238–246, New York, NY, USA, 1981. ACM.

[13] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[14] Erzsébet Németh and Tamás Bartha. Formal verification of safety functions by reinterpretation of Functional Block based specifications. In Darren Cofer and Alessandro Fantechi, editors, *Formal Methods for Industrial Critical Systems*, volume 5596 of *Lecture Notes in Computer Science*, pages 199–214. Springer Berlin / Heidelberg, 2009.

[15] Pataricza András, editor. *Formális módszerek az informatikában.* Typotex, 2. kiadás, 2005.

[16] Antti Valmari and Henri Hansen. Can stubborn sets be optimal? In Johan Lilius and Wojciech Penczek, editors, *Applications and Theory of Petri Nets*, volume 6128 of *Lecture Notes in Computer Science*, pages 43–62. Springer Berlin / Heidelberg, 2010.

[17] Harro Wimmel and Karsten Wolf. Applying cegar to the petri net state equation. *CoRR*, abs/1208.2159, 2012.