C:\Users\szarnyasg\Downloads\bme_logo_nagy.eps

**Budapesti Műszaki és Gazdaságtudományi Egyetem**

Villamosmérnöki és Informatikai Kar

Méréstechnika és Információs Rendszerek Tanszék

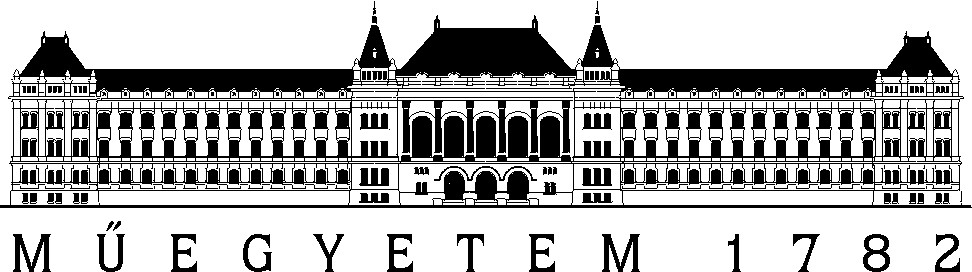
Petri-háló modellek részleges rendezés alapú verifikációja

Készítette

Kövér MártonKonzulens

Dr. Vörös András

2021



DIPLOMATERVEZÉSI FELADAT

**Kövér Márton**

Mérnökinformatikus hallgató részére

Petri háló modellek részleges rendezés alapú verifikációja

Aszinkron, elosztott és konkurens rendszerek modellezésére elterjedten használnak Petri hálókat, amelyek egyrészt grafikus reprezentációt nyújtanak, másrészt tulajdonságaik a matematika eszköztárát felhasználva formális módszerek segítségével vizsgálhatóak. Petri hálók esetén fontos kérdés az állapot elérhetőség, azonban ez a kérdés az állapottér hatalmas mérete miatt gyakran nehezen megválaszolható. Petri hálók esetén a nagy állapottér oka gyakran a lehetséges viselkedések különböző átlapolódásainak a nagy száma. Az irodalomban ezen probléma kezelésére a részleges rendezés alapú algoritmusokat fejlesztették ki.

A hallgató feladata megismerkedni az irodalomban fellelhető részleges rendezés alapú algoritmusokkal és ezek vizsgálata a Petri hálók formális analízisében.

A hallgató feladatának a következőkre kell kiterjednie:

* Végezzen irodalomkutatást a részleges rendezést használó algoritmusok területén, koncentrálva azon algoritmusokra, amelyek Petri hálókban is alkalmazhatóak.
* Vizsgálja meg az irodalomban elérhető fontosabb részleges rendezés alapú algoritmusokat Petri hálók analízisére. A vizsgálat alapján válasszon ki egy vagy több algoritmus, amelyet érdemes implementálni és megvizsgálni.
* Implementálja a kiválasztott részleges rendezés alapú technikákat.
* Hasonlítsa össze a megvalósított részleges rendezés algoritmusokat felderített állapotszám, a redukció mértéke, futásidő és tárigény szempontjából.
* Készítsen egy prototípus, amely vizualizációval szemlélteti az algoritmusok működését.

**Tanszéki konzulens:** Dr. Vörös András, adjunktus **Külső konzulens:**

Budapest, 2020.10.11.

…………………………..

Dr. Dabóczi Tamás

tanszékvezető

egyetemi tanár, DSc

Tartalomjegyzék

[Összefoglaló 6](#_Toc72259279)

[Abstract 7](#_Toc72259280)

[1. Bevezetés 8](#_Toc72259281)

[1.1. Motiváció 8](#_Toc72259282)

[1.2. Dolgozat felépítése 8](#_Toc72259283)

[2. Háttérismeretek 10](#_Toc72259284)

[2.1. Petri-hálók 10](#_Toc72259285)

[2.1.1. Felépítés 10](#_Toc72259286)

[2.1.2. Formális jelölések 11](#_Toc72259287)

[2.1.3. Dinamikus viselkedés 12](#_Toc72259288)

[2.1.4. Tüzelési szekvencia 12](#_Toc72259289)

[2.1.5. Szomszédossági mátrix 13](#_Toc72259290)

[2.1.6. Tüzelési invariáns 13](#_Toc72259291)

[2.1.7. Kiterjesztési lehetőségek 14](#_Toc72259292)

[2.1.8. Kifejező erő 15](#_Toc72259293)

[2.1.9. Elérhetőségi analízis 15](#_Toc72259294)

[2.2. Használt eszközök 17](#_Toc72259295)

[2.2.1. PetriDotNet 17](#_Toc72259296)

[2.2.2. Graphviz 19](#_Toc72259297)

[3. Részleges rendezés 22](#_Toc72259298)

[3.1. Teljes állapottér implementációja 22](#_Toc72259299)

[3.2. Stubborn sets 23](#_Toc72259300)

[3.2.1. Alap stubborn set módszer 24](#_Toc72259301)

[3.2.2. Statikus stubborn set definíciók 27](#_Toc72259302)

[4. Eredmények kiértékelése 32](#_Toc72259303)

[4.1. 8 bástya probléma 32](#_Toc72259304)

[4.2. DPhil 34](#_Toc72259305)

[4.3. FMS-15 35](#_Toc72259306)

[4.4. Hybrid-Cloud 37](#_Toc72259307)

[4.5. Kanban 38](#_Toc72259308)

[4.6. SlottedRing 40](#_Toc72259309)

[4.7. RoundRobin 41](#_Toc72259310)

[4.8. Phil 42](#_Toc72259311)

[4.9. Dfms 44](#_Toc72259312)

[4.10. Algoritmusok értékelése 47](#_Toc72259313)

[5. Működés bemutatása példákon 48](#_Toc72259314)

[6. Összefoglaló 52](#_Toc72259315)

[Irodalomjegyzék 53](#_Toc72259316)

[Függelék 55](#_Toc72259317)

Hallgatói nyilatkozat

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

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

Kelt: Budapest, 2021. 05. 23.

Kövér Márton

# Összefoglaló

A dolgozat célja, hogy a Méréstechnika és Információs Rendszerek Tanszéken fejlesztett PetriDotNet elnevezésű alkalmazáshoz egy plug-int készítsek. A PetriDotNet Petri-hálók szerkesztésére és működésének a szimulálásához készült. A plug-in feladata a Petri-hálóhoz tartozó teljes állapottér gráf generálása és annak csökkentése részleges rendezés alapú módszerekkel, mert a teljes állapottérgráf exponciálisan növekszik a háló bonyolultságával.

A dolgozat elkészítéséhez több módszer működésének is utána olvastam a szakirodalomban. A stubborn set módszert választottam ki, hogy implementáljam a plug-inhez. Ehhez megismertem két különböző módszert, illetve az egyikből kiindulva készítettem egy harmadik megoldást is.

Megvizsgáltam az elkészült algoritmusok működésének hatékonyságát, több a PetriDotNetben megvalósított modellen is. A modellek valós problémákon alapulnak és jelentős részük egy modellellenőrző versenyhez készült. Összeségében nem lehetett teljesen egyértelműen eldönteni, melyik algoritmus a leghatékonyabb.

Kisméretű Petri-hálókat felhasználva bemutattam, milyen kiindulási teljes állapottérgráfból, milyen részleges állapottér gráfokat lehet generálni az általam megvalósított algoritmusok segítségével.

# Abstract

The main goal of this thesis is to implement a plug-in for PetriDotNet tool. PetriDotNet is a tool to construct Petri nets and simulate their behaviour, it is developed by the Department of Measurement and Information Systems at BME. The task of this plug-in is to generate the exponentially growing full state space graph of the Petri nets and use partial order reduction methods to generate smaller state space graphs.

I read several papers in the topic of partial order reduction algorithms and I chose stubborn set methods for the plug-in. I use two different algorithms based on the papers and I implemented a third one with my improvements.

I examined the effectiveness of these algorithms on models included in PetriDotNet. The most of these models are based on real life problems from a model checking competition. In conclusion it could not be decided which algorithm performed the best, two of them have almost the same effectiveness depending on the examined model.

I presented on small Petri net models the result of the implemented methods using the generated state graphs.

# Bevezetés

A fejezet célja, hogy bemutassam a dolgozat megírásához a motivációimat, ismertessem a dolgozat célját és bemutassam a felépítését.

## Motiváció

Aszinkron, elosztott és konkurens rendszerek modellezésére elterjedten használnak Petri-hálókat, amelyek egyrészt grafikus reprezentációt nyújtanak, másrészt tulajdonságaik a matematika eszköztárát felhasználva formális módszerek segítségével vizsgálhatóak. Petri-hálók esetén fontos kérdés az állapot elérhetőség, azonban ez a kérdés az állapottér hatalmas mérete miatt gyakran nehezen megválaszolható. Petri-hálók esetén a nagy állapottér oka gyakran a lehetséges viselkedések különböző átlapolódásainak a nagy száma. Az irodalomban ezen probléma kezelésére a részleges rendezés alapú algoritmusokat fejlesztették ki.

A különböző matematikai jellegű problémák és azokra adott algoritmikus megoldások már régóta érdekelnek. Az egyetemi képzés során megismerkedtem részletesebben a gráfokkal és a Petri-hálókkal. Ezen a témához mindkettő matematikai ismerete elengedhetetlen, ezen felül új, érdekes algoritmusokkal is lehetőségem nyílt megismerkedni a dolgozat elkészítése során.

A feladat megvalósításhoz egy már létező, a Méréstechnika és Információs Rendszerek Tanszéken fejlesztett szoftvert fejlesztettem tovább. Célom volt, hogy akár tantárgyi keretek között is be lehessen mutatni a felhasznált algoritmusok működését, a kapott eredményeket.

## Dolgozat felépítése

A következő fejezetben bemutatom a dolgozat megértéséhez szükséges háttérismereteket, kezdve a Petri-hálókkal a felhasznált eszközökig. Az utána lévő fejezetben bemutatom részletesebben a teljes állapottér problémát, majd megoldást nyújtok rá részleges rendezés módszer és annak implementációjának bemutatásával. A stubborn set módszert választottam ki, és ehhez nyújtottam több megoldást is, így a kapott eredményeket nem csak a teljes állapottérhez lehet viszonyítani, hanem egymáshoz is.

A következő fejezetben példákon keresztül bemutatom az elkészült munkám hatékonyságát, vagyis a teljes állapottér bejárását és három stubborn set módszeren alapuló algoritmust. Majd kis Petri-hálókat felhasználva példaként bemutatom ezek működését és a kapott eredményket. Végül pedig összefoglalom a dolgozatban leírtakat.

# Háttérismeretek

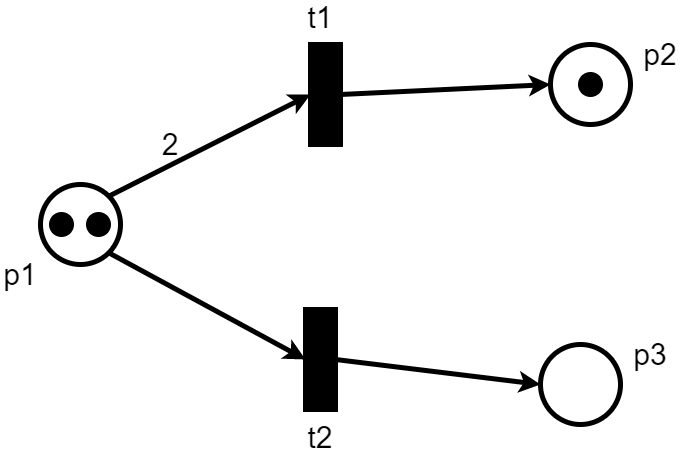
Ebben a fejezetben ismertetem a dolgozat megértéséhez szükséges alapismereteket, definíciókat. Bemutatom az egyszerű Petri-hálókat, azok lehetséges kiterjesztéseit és a hozzájuk tartozó kifejező erőt. A Petri-hálókkal kapcsolatban ismertetem, mi is annak az állapottere. Bemutatom a dolgozat készítése során általam használt eszközöket röviden.

## Petri-hálók

A Petri-hálókat[6][2] diszkrét dinamikus rendszerek működésének a leírására fejlesztették ki, eredetileg kémiai folyamatokhoz. Elterjedt alkalmazási köre a konkurens, aszinkron, elosztott, párhuzamos, nem determinisztikus rendszerek modellezése. A Petri- hálók előnye, hogy egyidejűleg biztosítanak valamilyen szintű grafikus reprezentációt és matematikai formalizmust is a modellhez. Könnyen kiterjeszthetők további elemekkel, melyek nagyban növelik a kifejező képességüket. Én a tiltó élek és a prioritás használatára térek ki.

### Felépítés

Az egyszerű Petri-hálók irányított, súlyozott páros gráfok. A csúcsok egyik osztálya a hely (**P**, az angol Place szóból), a másik pedig a tranzíció (**T**, az angol Transition szóból). Az élek (**E**, az angol Edge szóból) minden esetben egy és egy csúcsot kötnek össze, az él iránya tetszőleges lehet . A helyek segítsé-gével modellezhetők a különböző helyzetek, feltételek fennállása. Amennyiben fennállnak, akkor megjelöljük az adott helyet egy token segítségével, egy helyen tetszőleges számú token szerepelhet. A háló állapota leírható, ha megadjuk, hogy egy-egy adott helyhez mennyi token tartozik, ennek az elnevezése a tokeneloszlás. Grafikus reprezentációban a helyeket körök, a tranzíciókat téglalapok, az éleket nyilak, a tokeneket pedig a körökbe rajzolt pontok (ha nem fér ki, szám jelenik meg) jelölik. Bármely élhez rendelhető élsúly, mely azonos jelentéssel bír, mintha a két csúcs között számú párhuzamos él futna. Grafikusan nem párhuzamos éleket, hanem élsúlyokat használnak, az egyszeres élsúlyt nem szokás feltüntetni. Az 1. ábra egy egyszerű Petri-hálót ábrázol, melyen megfigyelhető, hogyan kell grafikusan megjeleníteni a helyeket, tranzíciókat, az éleket és a tokeneket.



1. ábra: Egyszerű Petri-háló élsúllyal

### Formális jelölések

Annak érdekében, hogy a Petri-hálok matematikai formában is leírhatók legyenek, formális leírást is érdemes bevezetni hozzájuk. Egy Petri-háló struktúrája az alábbi módon írható le: , ahol:

* a helyek véges halmaza
* a tranzíciók véges halmaza
* az élek halmaza
* a súlyfüggvény.

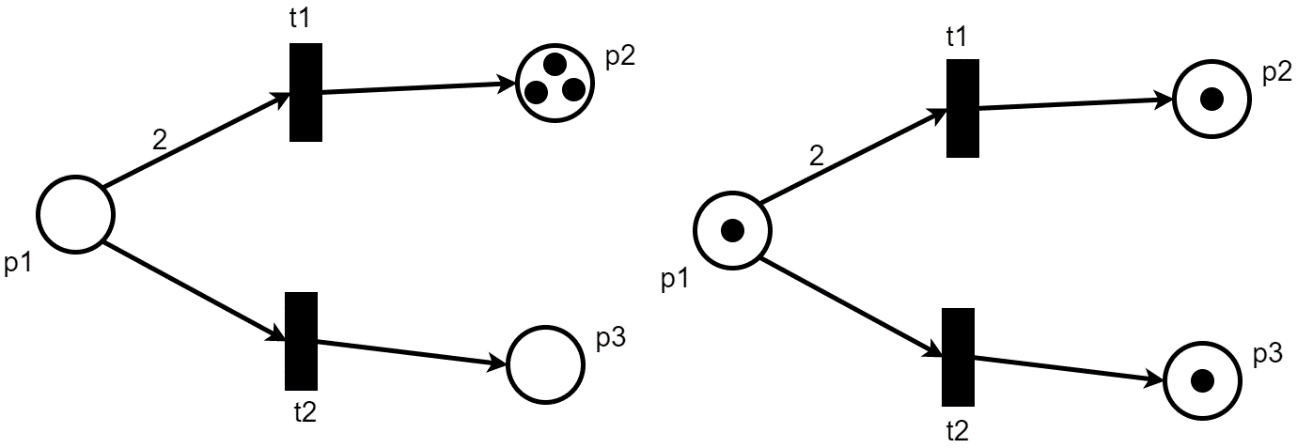
A tokenelosztást jelölhetjük egy vektor segítségével is, ha az állapotokat megszámozzuk, ez a tokeneloszlás vektor . Az 1. ábra által bemutatott Petri-háló esetében az . A fentebb struktúrára vonatkozó definíció kiegészíthető egy állapottal, mely a kezdeti tokeneloszlást adja meg .

Létezik formális jelölés a tranzíciók illetve a helyek kimeneteire és bemeneteire is, melyek az alábbiak:

* vagyis t bemeneti helyei
* vagyis t kimeneti helyei
* vagyis p bemeneti tranzíciói
* vagyis p kimeneti tranzíciói.

### Dinamikus viselkedés

A Petri-hálók működésének egy lépését, azaz az állapotváltozást, egy tranzíció tüzelésének nevezzük. Ennek a menete az, hogy kezdetben van egy *m* tokeneloszlás és van engedélyezett *t* tranzíció. Egy *t* tranzíció akkor engedélyezett, ha egy adott *m* tokeneloszlás mellett az összes bemeneti helyén van legalább annyi token, mint a hozzájuk tartozó élek súlya. Jelölése: . Formálisan ahol a *p*-ből *t*-be vezető él súlya. Következő lépésként az összes bementi helyről el kell venni tokent, majd az összes kimeneti helyre ki kell tenni (a él súlya) tokent. Egy engedélyezett tranzíció bármikor eltüzelhet, de nem kötelező tüzelnie. Ha több tranzíció is engedélyezett, akkor egyszerre mindig csak egy tüzelhet, ami véletlen választással dől el. Ezáltal nemdeterminisztikusan működnek a Petri-hálók.



2. ábra: A 1. ábra által ábrázolt háló a lehetséges tüzelések után

A 2. ábra azt mutatja be, milyen új állapotokat vehet fel az 1. ábra által bemutatott Petri-háló tüzelés után. Látszik, hogy két különböző tranzíció is engedélyezett volt, amik közül véletlenszerűen tüzel az egyik, ezért két különböző új állapotot is létrejöhet. Az is jól látszik, hogy az ábrán szereplő *t1* tranzíció tüzelése kizárja a *t2* tüzelését és viszont, ellenben a *t2* tüzelhet egymás után kétszer is.

### Tüzelési szekvencia

Ha a kiinduló állapot *M*, és *t* tüzel, akkor az új *M’* állapot, akkor azt jelöli. Tüzelési szekvenciának nevezzük a vektort, mely tüzelések egymásutániságát adja meg.

vagy

Ha közül az összes kielégíti egymás után a tüzelési szabályt és *m*-ből elérhet *m’*  által, akkor azt módon jelöljük. Azt, hogy egy tranzíció hányszor tüzelt egy tüzelési szekvenciában, a tüzelési szám vektor adja meg ().

### Szomszédossági mátrix

A súlyozott szomszédossági mátrixot az élsúlyok alapján lehet felvenni, a *W*  mátrix dimenziói lesznek. A *w(t,p)* megadja, hogy *t* tüzelése esetén mennyit változik a *p*-beli tokenszám.

A definícióból is látszik, hogy a szomszédossági mátrix nem írja le egyértelműen a Petri-hálót. Ha egy hely és tranzíció között fut él mindkét irányba, a mátrixba 0 érték kerül, viszont a tranzíció nem lesz engedélyezett, ha a helyen 0 token van. Érdemes lehet a mátrixokat használni, mert akkor nincs ilyen probléma.

Az 1. ábra Petri-hálójához tartozó súlyozott szomszédossági mátrixok:

Ha a kiinduló állapot *M*, és *t* tüzel, akkor az új *M’* állapot leírható az alábbi módon:

### Tüzelési invariáns

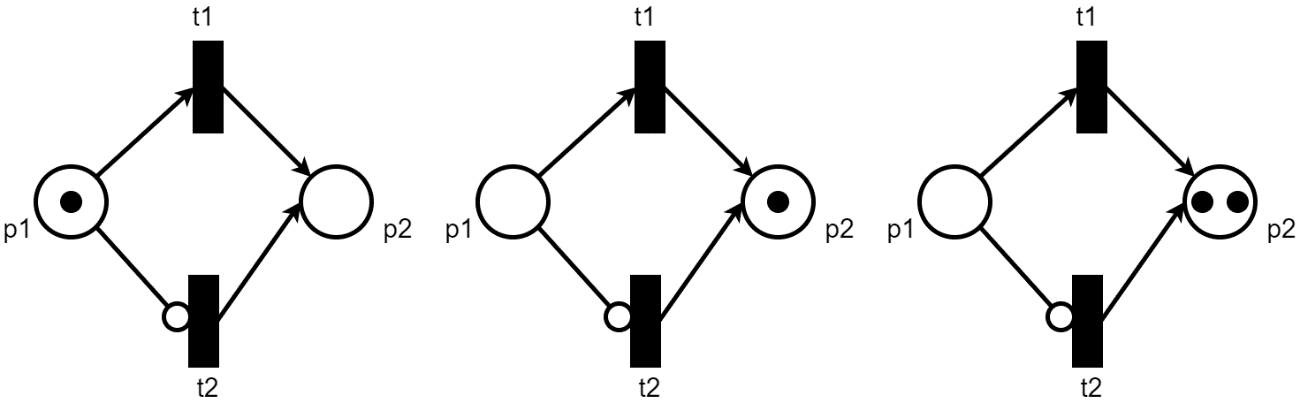
A tüzelési szám vektor tüzelési invariáns (T-invariáns), ha végrehajtása nem változtatja meg a tokeneloszlást. Ez azt jelenti, hogy ciklus van az állapottérben, . Ilyenkor . Az előző egyenlet megoldásánál érdemes megjegyezni, hogy megoldások többszöröse is megoldás, megoldások összege is megoldás és a megoldások lineáris kombinációja is megoldás. Egy T-invariáns realizálható, ha létezik hozzá végrehajtható tüzelési sorozat.

### Kiterjesztési lehetőségek

Az egyszerű Petri-hálók többféleképpen is kiterjeszthetők a modellezési erő növelése vagy a működés nemdeterminisztikus voltának csökkentése érdekében. Ilyenek például a kapacitás korlát bevezetése, azaz csak adott számú token lehet egy helyen, a tiltó élek használata és a prioritás bevezetése a tranzíciókhoz. Az utóbbi két módszerről részletesebben is írok.

#### Tiltó él

Tiltó él esetén fordítottan működik az engedélyezettség feltétele, vagyis ahelyett, hogy elvárnánk egy feltételt, azt várjuk, hogy ne teljesüljön. Grafikusan a tiltó élet egy kör jelöli az él végén nyíl helyett. Tiltó él esetén a tüzelési szabály a következő lesz: Ha a *t* tranzícióhoz kapcsolódó bármely *(p,t)* tiltó él bemenő helyén élsúlynál nagyobb vagy egyenlő számú token van, akkor a tüzelés nem hajtható végre.



3. ábra: Példa a tiltó éles Petri-háló működésére

A 3. ábra egy példát mutat be a tiltó élek használatára. Az ábrán szereplő hálók közül a baloldalon szereplő az első, a középső a *t1* tranzíció tüzelésével kapható, a jobboldali háló pedig a középsőhöz képest a *t2* tüzelésével.

#### Prioritás

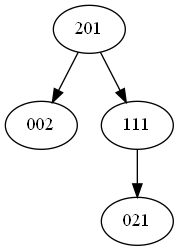
A prioritás bevezetésével megoldható, hogy a nemdeterminisztikus tüzelés helyett egy prioritási sorrendben tüzeljenek az engedélyezett tranzíciók. A tüzelési szabály itt annyiban módosul, hogy egy alacsonyabb prioritású engedélyezett tranzíció addig nem tüzelhet, amíg van nála magasabb prioritású és engedélyezett tranzíció. Ha azonos a prioritása a tranzícióknak, továbbra is nemdeterminisztikus viselkedés várható.

### Kifejező erő

Mind a tiltó él használata és mind a prioritás bevezetése is növeli a Petri-háló kifejező erejét, ugyanis képessé teszik azt „zero testingre”. A „zero testing” lényege, hogy eldönthető-e valamilyen formában az, hogy egy adott állapotban van-e token. Ha ez eldönthető, akkor lehet általánosítani és akármekkora korlátos szám esetén ellenőrizni, annyi token van-e adott helyen. Ez a képesség a Turing gépekkel azonos kifejezőerőjűvé teszi a tiltó éles és prioritásos Petri-hálókat. Az egyszerű Petri-hálók nem képesek ennek a vizsgálatára, így kifejezőképességük alacsonyabb.

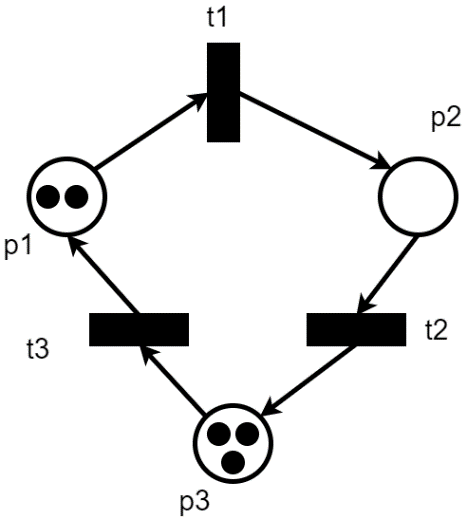
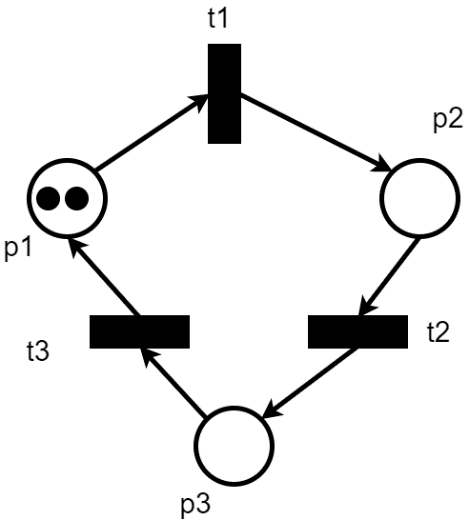
### Elérhetőségi analízis

Az állapottér a kezdőállapottól függő viselkedés leírására használható. Azokat az állapotokat tartalmazza, melyek elérhetők a kiinduló állapotból, formálisan . A teljes állapottér leírására az elérhetőségi gráfot használjuk. Egy Petri-hálóhoz tartozó teljes elérhetőségi gráf egy irányított páros , ahol (az elérhető állapotok halmaza) a csúcsok halmaza, az élek halmaza. Az állapottér a Petri-háló méretével exponenciális mértékben is nőhet, így az könnyen kezelhetetlenné válhat.

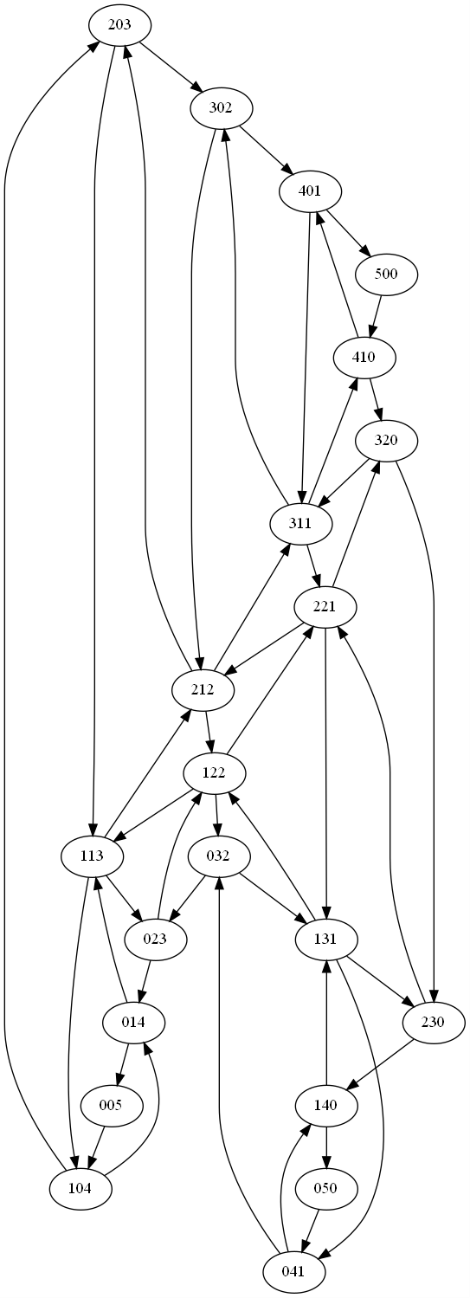
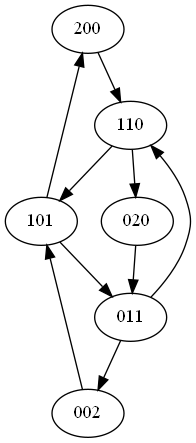


4. ábra: A 1. ábra által mutatott Petri-hálóhoz tartatozó teljes elérhetőségi gráf

Az elérhetőségi gráf csúcsaiban található értékek azt mutatják, hogy sorra hány tokent tartalmaz adott állapotban a hálóban egy-egy hely.



5. ábra: Példa Petri-hálók



6. ábra: A 5. ábra szereplő Petri-hálók teljes elérhetőségi gráfja

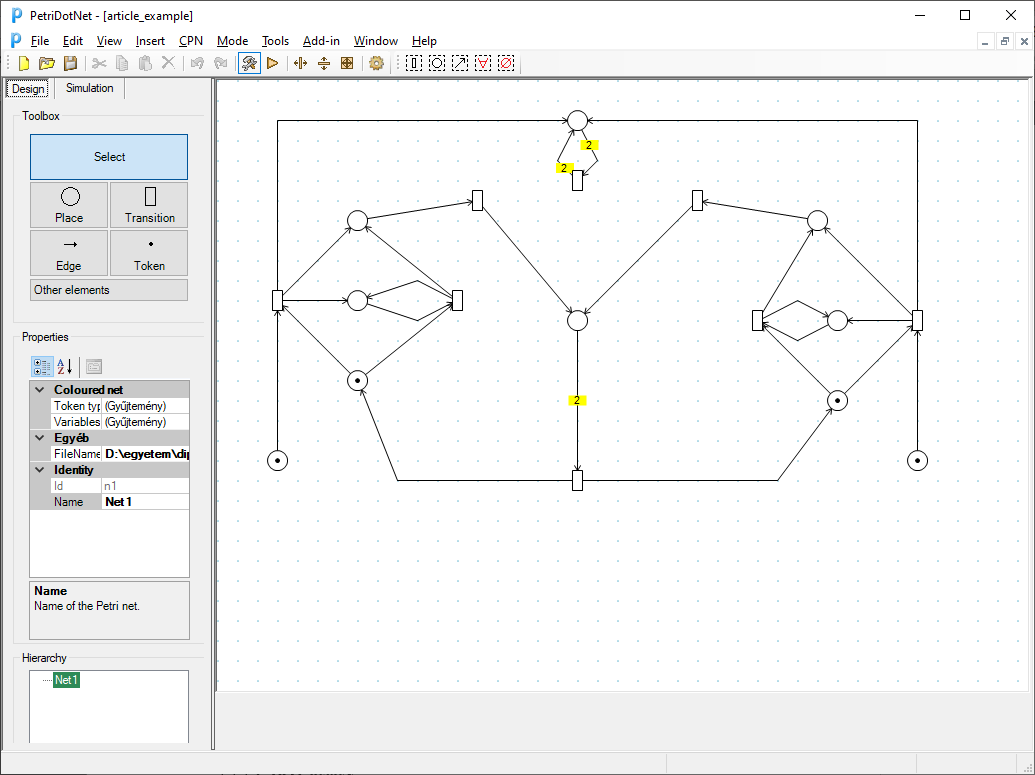
A 4. ábra, 5. ábra, 6. ábra azt mutatja be, milyen gyorsasággal növekszik a teljes elérhetőségi gráf. A 4. ábra az 1. ábra által bemutatott Petri-hálóhoz tartozik. Látható, hogy csak négy csúcsból áll és mivel nincs ciklus a hálóban, ezért a gráf se tartalmaz kört. Az 5. ábra és 6. ábra azt mutatják be, hogy mekkora különbséget jelenthet az elérhető állapotok között azonos struktúrájú Petri-háló esetén is a különböző kiindulási állapot. A baloldali gráf tartozik a baloldali hálóhoz.

## Használt eszközök

Ebben a fejezetben bemutatom az elkészített munkámhoz használt eszközöket. A fejlesztést C# nyelven végeztem és Microsoft Visual Studio 2019-et használtam. Ezeket nem mutatom be részletesebben, mert manapság elterjedt a használatuk. Általam használt eszközök közül a PetriDotNet és a Graphviz működését mutatom be.

### PetriDotNet

A PetriDotNet[5] egy eszköz Petri-hálok szerkesztésére, szimulálására és analizálására. A Budapesti Műszaki és Gazdaságtudományi Egyetemhez tartozó Hibatűrő Rendszerek Kutatócsoport készítette, úgy, hogy könnyen használható és bővíthető legyen. Az eszköz néhány száz helyből álló hálók szerkesztését és szimulációját támogatja, de néhány ezer helyből álló hálókon is futtathatók analízisek. Az eszköz képes kezelni hierarchia szinteket és színes Petri-hálókat is. A szimulációnál beállítható automatikus léptetés és manuális léptetés is. Az háló analíziseket plug-inek segítségével végezhetjük el.



7. ábra: PetriDotNet fő ablaka egy betöltött Petri-hálóval

#### Plug-in fejlesztés

A PetriDotNet funkcionalitása bővítmények (plug-inok) segítségével bővíthető, melyek rendelkezhetnek szimulációs feladattal, biztosíthatnak analitikus funkciókat vagy bővíthetik az importálás/exportálás funkciókat. A bővítmények hozzáférnek a Petri-háló adatmodelljéhez, használhatják a grafikus felhasználói felületet (GUI), új menü elemet hozhatnak létre és PetriDotNet parancsokat is végrehajthatnak.

Az összes bővítményt .NET osztálykönyvtárként (dll) lehet hozzáadni az alkalmazáshoz, az osztályok valamelyikének implementálnia kell a IPDNPlugIn interfészt, melyhez különböző annotációk is tartoznak. Ez az interfész egy Initialize eljárást tartalmaz, ebben lehet lekérni az appDesc-et, amin keresztül interakcióba lehet lépni a programmal. Ilyen interakció lehet a Petri-háló elkérése és különböző eljárások hívása, például mentés, betöltés. Az Initialize eljárásban kell továbbá felvenni az új menü elemeket is, melyeken keresztül elérhetővé válnak a bővítmény funkciói. Az új menü elemek az Add-in menüpont alá kerülnek.

A dolgozatom témáját egy bővítmény fejlesztésével valósítottam meg, ami több különböző funkcióval rendelkezik. Ezekről még írok részletesebben a dolgozat 3. fejezetében.

Tekintve, hogy a fejlesztés eredménye (dll) önállóan nem futtatható, a készült kód debugolásához meg kell adni különböző beállításokat a Visual Studio számára. Az első ilyen a Debug típusú fordítás kimeneti helyének megadása. Mivel a PetriDotNet csak a futtatható állománnyal egy könyvtárban található add-in könyvtárban lévő dll fájlokat használja fel, ezért ezt kell megadni célkönyvtárnak. Ezután le kell fordítani a kódot. Miután elindult a PetriDotNet, Visual Studioban megadható a Debug menüből az Attach to Process almenüt kiválasztva, mely futó folyamathoz kívánjuk hozzácsatolni a kódot. A beállítások elvégzése után a szokásos debug funkciók elérhetők.

### Graphviz

A Graphviz[7] egy nyíltforráskódú gráfok vizualizációját végző szoftver. A gráfok vizuális megjelenítése a tudomány több területén is fontos, ilyenek például a számítógépes hálózatok, machine learning, bioinformatika, adatbázisok stb. A Graphiz elrendezést végző programja gráfok szöveges leírásából készít diagrammokat több különböző formátumban. Ilyenek lehetnek az svg weblapokhoz, PDF dokumentumokhoz, képformátumok és más egyebek. Számos hasznos funkcióval rendelkezik a szoftver a diagrammok készítése során, állíthatók a színek, betűtípusok, vonalstílusok, fel lehet venni saját alakzatokat stb.

A képen szöveg látható

Automatikusan generált leírás

8. ábra: A Graphviz logoja

Én a Graphvizt arra használom, hogy a felépített állapottér gráfjaimat meg tudjam jeleníteni megfelelően. Ezeket használom fel példának, így ebben a dolgozatban látható gráfokat is ennek a segítségével készítettem. Az elkészült bővítményem kimeneteként megkapható egy DOT nyelven írt szöveges fájl, amit a Graphviz parancssoros környezetének segítségével alakítottam át a kívánt képi formátumra.

#### DOT nyelv

A DOT nyelv gráfok leírására szolgáló szöveges nyelv. Feldolgozását több különböző program is végezheti, de a céljuk a grafikus formában való megjelenítés. Működését legegyszerűbb egy példával bemutatni.

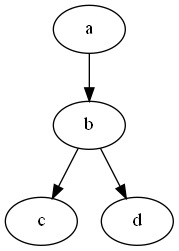
digraph graphname {

a -> b -> c;

b -> d;

}

1. kódrészlet: Irányított gráf DOT nyelven



9. ábra: az 1. kódrészlethez tartozó gráf

Az 1. kódrészlet és az ahhoz tartozó 9. ábra egy egyszerű irányított gráfot mutatnak be példaként. A szöveges leírásban a digraph kulcsszó jelöli, hogy egy irányított gráfról van szó, ezután meg kell adni a gráf nevét, a csúcsok közötti kapcsolatot egy nyíllal kell jelölni, mert irányított a gráf, ellenkező esetben „--„ lenne a megfelelő jelölés.

graph graphname {

// ez a tulajdonság a gráfra vonatkozik

size="4,4";

# A label tulajdonság segítségével állítható egy csúcs neve

a [label="Foo"];

// A csúcs formája is állítható a shape tulajdonsággal

b [shape=box];

/\* Állíthatók az élek tulajdonságai is

például a szín, az éltípus és az elnevezés

\*/

a -- b -- c [color=blue];

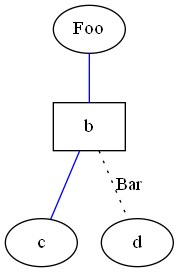
b -- d [style=dotted, label="Bar"];

// [style=invis] elrejt egy csúcsot.

}

2. kódrészlet: Tulajdonságokkal ellátott gráf DOT nyelven

A 2. kódrészlet és az ahhoz tartozó 10. ábra, azt mutatják be, milyen különböző tulajdonságai állíthatók egy-egy gráfnak. Vannak általános tulajdonságok, amik a teljes gráfra vonatkoznak, vannak, amik élekre, és vannak, amik csúcsokra. DOT nyelven lehet kommenteket is tenni a gráf leírásába, többféle módon is. Ahogy a 2. kódrészlet is mutatja: egysoros kommentek hagyhatók a „//” jelzés vagy a „#” jelzés után. Több soros kommenteket „/\* <komment>\*/” jelölésben a <komment> helyére kell beilleszteni. Az itt felsoroltakon kívül még további tulajdonságok is állíthatók, például lehet beállítani részgráfokat is, melyeknek lehet több különböző szerepe, de ezeket nem használtam.



10. ábra: A 2. kódrészlethez tartozó gráf

A DOT nyelv feldolgozására a Graphviz által tartalmazott dot.exe eszközt használtam, mely parancssoros környezetből futtatható, működése különböző kapcsolók állításával módosítható.

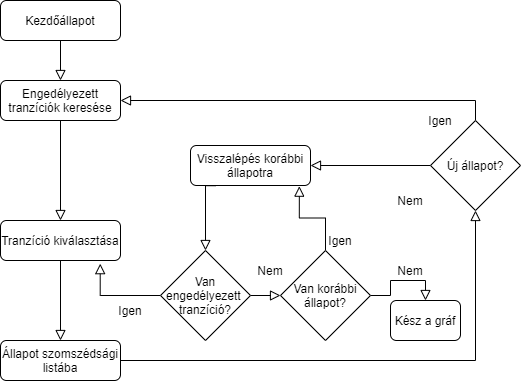
# Részleges rendezés

Annak érdekében, hogy az állapottér kezelhető méretű maradjon nagyobb méretű Petri-hálók esetén is, a teljes állapottér feltérképezése helyett egy úgynevezett részleges rendezést használó algoritmus segítségével csökkenteni tudjuk a kapott gráf méretét. Ebben a fejezetben ilyen módszereket fogok bemutatni, illetve a teljes állapottér feltérképezéséhez használt kódot, mert azt használtam kiindulásnak a továbbiakhoz.

## Teljes állapottér implementációja

A PetriDotNet saját típust használ a Petri-hálók tárolására, feldolgozására, amit én átalakítottam annak érdekében, hogy az én igényeimnek is megfeleljenek. A háló tulajdonságai közül nem tartottam meg mindet, egyedül azt, hogy hány token van az adott helyeken, illetve, hogy a tranzíciókhoz tartozó élek hány tokent adnak vagy vesznek el. A tranzíciókhoz két vektor tartozik a megoldásomban, az egyik a tranzícióhoz szükséges tokenek számát tárolja helyenként, a másik pedig azt, hogy hova kell és mennyi új tokent kitenni. Azért emellett a megoldás mellett döntöttem, mert így eldönthető, hogy tüzelhet-e olyan tranzíció is, ami egy adott helyre ugyanannyi tokent tesz ki, mint amennyit elvesz. A haló állapotát ugyancsak vektor formájában tároltam el, jelölve adott helyhez mennyi token tartozik. Egy adott állapot meg tudja adni, hogy egy tranzíció tüzelhet-e belőle, és lehet tüzelni is tranzíciót. Új állapot létrehozható meglévő állapot és egy tranzíció segítségével is.

Annak érdekében, hogy nagyobb hálók feldolgozása esetén se fagyjon le a PetriDotNet GUI-ja, külön szálon végzem a számításokat, mely jelzi a felhasználó számára, ha elkészült. A teljes állapottér felderítését mélységi bejárás segítségél végeztem. Minden állapothoz feljegyeztem, hogy mely tranzíciók tüzelhetnek belőle. Az első tranzíció tüzelése után feljegyeztem, hogy a kapott új állapot szerepelt-e már korábban, ha igen, akkor csak hozzáadtam a kiinduló állapothoz, mint szomszéd, különben megvizsgáltam az ő esetében is az elérhető tranzíciókat. Addig haladtam előre új állapotok mentén, míg egy állapotból nem volt már több tüzelhető tranzíció, ekkor egyet visszalépve az ottani állapotból próbáltam továbblépni. A 11. ábra folyamatábraként mutatja, be az elérhetőségi gráf készítésének módját.



11. ábra: A teljes elérhetőségi gráfkészítés algoritmusának folyamatábrája

Ha a kapott gráf mérete kezelhető, akkor DOT nyelven mentésre kerül fájlba, melyet később fel lehet dolgozni. Túl nagy teljes állapottér esetén csak az állapotok számát írom ki, az eredmény nem lenne úgysem áttekinthető.

## Stubborn sets

Az állapottér készítés hasonlóan működik stubborn setek[4][3][1][8] használata esetén is, mint ha teljes állapotteret vizsgálnánk egy Petri-hálóhoz. A különbség annyi, hogy ahelyett, hogy adott állapothoz az engedélyezett tranzíciókat vizsgálnánk, egy úgynevezett stubborn setet kell meghatározni. A stubborn set engedélyezett elemei alapján kell meghatározni a következő állapotokat. A kimaradó engedélyezett tranzíciók vagy később kerülnek valamilyen formában feldolgozásra, vagy egyáltalán nem fontosok az analízis eredménye szempontjából. Mivel nem minden engedélyezett tranzíciót vizsgálunk meg egy-egy állapotból, ezért egyre kevesebb lesz az elérhető új állapotok száma. A kapott gráf a teljes elérhetőségi gráfnak egy részgráfja lesz, a részleges elérhetőségi gráf.

A stubborn setek készítése két dologtól függ, a tranzíciók közötti összefüggésektől és attól, milyen tulajdonságokat kívánunk vizsgálni. Úgy kell kiválasztani, milyen stubborn set módszer alkalmazzunk, hogy az eredményben szereplő tranzíciók és állapotok még tartalmazzák a vizsgálni kívánt tulajdonságokat, de a lehető legkisebb legyen a kapott gráf.

### Alap stubborn set módszer

A D1 és D2 feltételek használatával megőrizhetők a holtpontok, vagyis azok az állapotok, melyek nem rendelkeznek engedélyezett tranzícióval. Tranzíciók egy halmaza akkor és csak akkor nevezhető **dinamikusan** „stubborn-nak” -hoz, ha az alábbi két feltétel teljesül:

* **D1** Ha , , és , akkor létezik állapot, hogy és .
* **D2** Ha -nak van engedélyezett tranzíciója, akkor van legalább egy , amelyre teljesül: ha és , akkor állapotban engedélyezve van. Minden ilyen tulajdonságú tranzíciót a stubborn set egy kulcstranzíciójának nevezünk.

**Ha az *n = 0 D2*-ben, akkor látható, hogy a kulcstranzíció engedélyezett. Egy tranzícióhalmaz erősen dinamikusan stubborn *M* állapotban, akkor és csak akkor, ha dinamikusan stubborn halmazt alkotnak *M*-ben és minden engedélyezett tranzíció kulcstranzíció, vagyis igaz rájuk a *D2* szerint. Azért lényeges megemlíteni az erősen dinamikus alkategóriáját a dinamikus stubborn seteknek, mert sok stubborn set algoritmus ilyeneket állít elő, illetve néhány analízist végző algoritmus használatához szükségesek.**

**Tekintve, hogy a dinamikus stubborn seteknek tartalmazniuk kell kulcstranzíciót, a holtpontokhoz nem tartozik dinamikus stubborn set. Ha *M* holpont, akkor a tranzíciók teljes halmaza dinamikusan stubborn, sőt még erősen dinamikusan is stubborn *M*-re.**

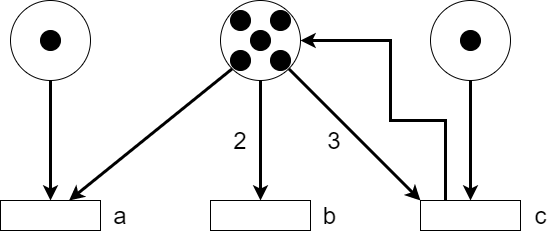
**Legyen egy olyan függvény, ami minden nem holtpont *M* állapothoz hozzárendel egy dinamikus stubborn setet. A fent bemutatott alap stubborn set módszer a kezdő állapotból indulva minden megtalált *M* állapothoz megadja azokat a kimenő éleket és következő állapotokat, melyek megkaphatók a -ben lévő engedélyezett tranzíciók tüzelésével.**

**Sok különböző definíció fellelhető a stubborn setekhez, melyek jellemzően különböző erősségűek. Akkor mondhatjuk, hogy egy stubborn set definíció gyengébb, mint egy másik, ha minden halmaz, ami stubborn a második szerint, az előbbi szerint is az, de ez nem feltétlen kölcsönös módon igaz.**

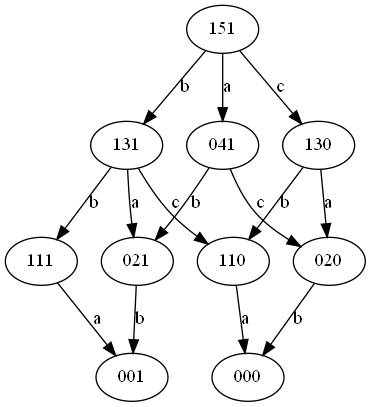
**Minél gyengébb egy stubborn set definíció, annál több halmazt fogad el, így nagyobb lehet az esélye olyan halmaz megtalálásának, ami csak nagyon kevés tranzíciót tartalmaz. Ennek következményeképpen a gyengébb definíciók nagyobb potenciállal rendelkeznek jobb redukciós eredmények elérésére, mint az erősebbek.**

#### ****Példa a dinamikus stubborn set módszer működésére****

A dinamikus stubborn set módszer alkalmazásához szükséges a Petri-háló későbbi állapotainak is az ismerete. A 12. ábra egy Petri-hálót ábrázol, melyen bemutatom a fent ismertetett algoritmus működését, a 13. ábra pedig a hozzátartozó teljes állapotteret ábrázolja.



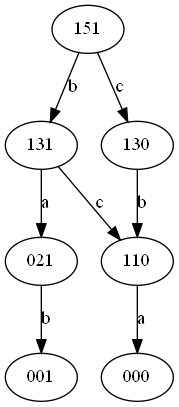
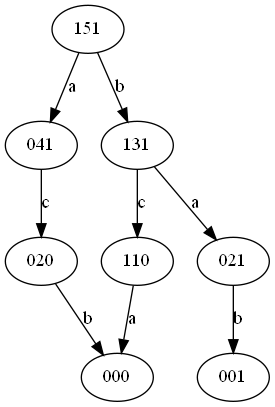
12. ábra: Példa Petri-háló[3]



13. ábra: A 12. ábra által mutatott Petri-hálóhoz tartozó teljes állapottér

Első lépésként a kiinduló állapothoz kell megkeresni a megfelelő stubborn setet. Itt mindhárom ***(a,b,c)*** tranzíció tüzelhet, így meg kell őket vizsgálni. A ***c*** és ***ac*** tranzíciók halmaza nem jó megoldás, mert ***bb*** tüzelési szekvencia végrehajtása után ***c*** már nem tüzelhet, nem teljesül a ***D2*** feltétel. Az *a* tranzíció se jó megoldás, mert ***bca*** tüzelési szekvencia található a teljes állapottér gráfban, de ***abc*** nem, ez sérti a ***D1*** feltételt. A ***b*** esetén ugyancsak a ***D1*** feltétel nem teljesül, ***acb*** található a gráfban, de ***bac*** nem. A minimális megoldás a ***{b,c}*** és a {***a,b}*** lesznek. Ez mutatja azt is, hogy a minimális stubborn setek meghatározása nem egyértelmű adott állapotban. A 14. ábra bemutatja mindkét kiindulási halmazhoz tartozó eredményt, de csak a ***{b,c}***-hez tartozót vezetem le teljesen. A következő vizsgált csúcs a gráfban a ***131*** jelzésű. Nem lesz jó megoldás ***a***, mert ***ca*** szekvencia létezik, ***ac*** pedig nem (***D1*** feltétel), ***b*** nem megfelelő, mert *c* tüzelése letiltja (***D2*** feltétel), ***c*** pedig nem megfelelő, mert ***ab*** és ***ba*** szekvencia tüzelése is letiltja (***D2*** feltétel). A minimális halmaz az ***{a,c}*** lesz. A ***130*** jelzésű csúcshoz a ***{b}*** halmazt használtam minimálisként, de az ***{a}*** is jó megoldás lenne. Ehhez azonban már nem készítettem külön ábrát. A ***021*** és a ***110*** jelzésű csúcsoknak is csupán egy-egy engedélyezett tranzíciója van, ezek megfelelő stubborn setek is lesznek. A ***001*** és ***000*** jelzésű csúcsok pedig holtpontok lesznek a Petri-hálóban, így belőlük már nincs engedélyezett tranzíció.

A teljes állapottérgráf 10 csúcsot tartalmaz, a dinamikus stubborn módszer segítségével ez lecsökkenthető akár 7-re is. Mint a 14. ábra is mutatja, nem mindegy az, hogy mit választunk minimális halmaznak adott lépésben, még akkor sem, ha azok elemszáma megegyezik, viszont ez nem derül ki csak későbbi lépések során, mikor már a teljes eredmény rendelkezésre áll. A bal oldali gráf 8, míg a jobb oldali 7 csúcsot tartalmaz.



14. ábra A 12. ábra által mutatott Petri-hálóhoz tartozó állapotterek alkalmazva a dinamikus stubborn set módszert

### ****Statikus stubborn set definíciók****

Annak érdekében, hogy az alap stubborn set módszer használható legyen a gyakorlatban is, szükség van olyan algoritmusra, ami a nem holtpont állapotokból megadja a dinamikus stubborn set definíciónak megfelelő engedélyezett tranzíciók halmazát. Az összes engedélyezett tranzíciót tartalmazó halmaz is stubborn set a nem holtpont állapotokban, azonban ez nem hatékony, mert nem csökkenti az adott állapothoz tartozó kimenő élek számát, ezért csak akkor érdemes használni, ha nincs jobb megoldás. A dinamikus stubborn set módszer nem vezet használható algoritmusra, ugyanis olyan állapotokat is felhasznál, amik csak később kerülnek előállításra.

Ennek a problémának a megoldására szolgál a statikus stubborn set bevezetése, melynek megadása függ attól is, formálisan hogyan van definiálva a vizsgálandó modell. Több különböző definicíó is létezik ez alapján, melyek hatékonysága változik attól is függően, mennyi energia van fektetve a tranzíciók közötti függőségek vizsgálatára. A lényeges ezen algoritmusok esetében, hogy a kapott stubborn setek bizonyíthatóan megfeleljenek a dinamikus stubborn set feltételeinek is.

#### Statikus definíció 1

Az alábbi feltételek egy egyszerű definíciót adnak statikus stubborn set keresésére, mely algoritmikusan is megvalósítható:

1. Ha , akkor .
2. Ha , akkor .
3. Ha , akkor

A fenti definíció *D1* és *D2* feltételeknek megfelelő stubborn setet generál, továbbá az eredmény erősen dinamikus stubborn set kategóriába is beleesik.

Az első feltétel azt mondja ki, ha létezik engedélyezett tranzíció *M*-ben, akkor a stubborn setben is kell lennie ilyennek. A második szabály feladata biztosítani, ha egy nem engedélyezett *t* tranzíció is bekerül a stubborn setbe, akkor az ne válhasson engedélyezetté a stubborn seten kívüli tranzíciókat is tartalmazó tüzelési szekvencia hatására sem (*D1* feltétel első fele). Ezt úgy éri el, hogy kiválaszt egy *p* helyet, ahol nincs elegendő token ahhoz, hogy a jelenlegi *M* állapotban *t* engedélyezett lehessen és *p* minden bemenő tranzícióját felveszi a stubborn setbe. Azon tranzíciók számítanak bemenőnek *p* szempontjából, amik hozzáadhatnak tokent, vagyis amik *t*-t engedélyezetté tudják tenni. A harmadik feltétel feladata pedig, hogy a stubborn setben lévő engedélyezett tranzíciókat ne tudja letiltani olyan tranzíció, mely nem része annak (*D2* feltétel). Ezt úgy éri el, hogy minden tranzíciót felvesz a stubborn setbe, amelyek tokent távolítanak el a vizsgált tranzíciók bármely bemeneti helyéről. Az így kapott eredményhalmazra igaz lesz, hogy minden stubborn setbeli tranzíció kulcstranzíció is egyben. Annak érdekében, hogy *D1* második felének igazsága is meg legyen mutatva, még látni kell, hogy a harmadik feltétel biztosítja azt is, hogy stubborn set által tartalmazott tranzíciók tüzelése nem fogja letiltani az azon kívül eső tranzíciók tüzelését.

Példákat a működésre későbbi fejezetben fogok bemutatni, de az algoritmus hatékonyság szempontjából nem lesz maximális, vagyis nem minden esetben a minimális méretű stubborn setet fogja megadni egy-egy állapothoz.

##### Implementáció

A hatékonyság növelése érdekében a stubborn set építése során kiindulásként használt tranzíciót nem véletlenszerűen választottam meg, ahogy a fenti szabályok azt megengednék, hanem megkerestem azt az engedélyezett tranzíciót, melynek tüzelését a jelenlegi állapotból kiindulva a legkevesebb tranzíció tüzelése tudja közvetlenül letiltani. Az így kapott tranzíciót egy listába tettem, melynek összes elemére alkalmaztam a fenti kettes és hármas szabályt. A listát természetesen folyamatosan bővítettem az új kapott tranzíciókkal. Az összes elem vizsgálata után a listám tartalmazza a stubborn setet. A kettes és hármas feltételek alkalmazásában nincs lényegi eltérés, csak az általam használt adatszerkezetekhez igazítottam őket.

#### Statikus definíció 2

A következő definíció összetettebb az előzőnél, de ugyancsak következik belőle *D1* és *D2*. Ebben az esetben részletesebben van vizsgálva a tranzíciók közötti függőség. Ez a definíció következik az előzőből, de nem egyenértékű azzal. Ebből következően több halmazt is elfogadhat stubbornak vagyis kisebb stubborn seteket produkálhat, ami javítja a redukciós képességét.

1. Ha -ből nem engedélyezett *t*, akkor létezik melyre és .
2. Ha -ből engedélyezett *t*, akkor minden esetén vagy .
3. Létezik legalább egy , amire *M*-ből engedélyezett , és minden helyre: .

A definíció első feltétele (*F1*) azt adja meg, mikor kell felvenni a stubborn setbe egy nem engedélyezett *t* tranzícióhoz további tranzíciókat, a második (*F2*) pedig engedélyezettekhez mondja ki ugyanezt. A harmadik feltétel feladata a kulcstranzíciók meghatározása.

##### Implementáció

Az implementációt a [2] forrásban szereplő Algoritmus 3 alapján készítettem kis módosítással, mert egy helyen szerintem hibásan szerepelt az algoritmus.

**bemenet**: *(N,m)* Petri-háló, *T* a tranzakciók halmaza

**kimenet**: stubborn set

**for** minden **do**

|

| **for** minden és minden **do**

| | **if** **then**

| | ;

| | **end**

| **end**

| **if** **then**

| |

| |

| **end**

**end**

**while** amire F1 vagy F2 nem teljesül **do**

**end**

3. kódrészlet: Statikus algoritmus 2

#### Statikus definíció 3

Az előző fejezetben bemutatott statikus stubborn set módszert kipróbálva láttam lehetőséget arra, hogy fejlesszem az algoritmus hatékonyságát. A [8] cikk felhasználásával az alábbira módosítottam a korábban bemutatott három szabályt:

1. Ha -ből nem engedélyezett *t*, akkor létezik melyre és .
2. Ha -ből engedélyezett *t*, akkor minden esetén és .
3. Létezik legalább egy , amire *M*-ből engedélyezett , és minden helyre: .

Az így kapott szabályrendszer erősen dinamikus stubborn seteket ad eredményül, ugyanis a tüzelőképes tranzíciókhoz bekerült egy új kitétel, mely biztosítja, hogy kulcstranzíciók lesznek.

A további módosításokat végeztem még az algoritmuson. Az első ilyen, hogy mikor keresem a kiindulásként felhasznált kulcstranzíciót, azt eddig is úgy tettem, hogy vizsgáltam, hány másik tranzíciót kéne hozzáadni a stubborn sethez és ebből kiválasztottam a legkisebb halmazt. A változtatás lényege, hogy nem csak az így kapott tranzíciót használom fel kiindulásként, hanem a teljes halmazt. A második változtatás pedig, hogy ha egy tranzíció bekerült már a stubborn setbe, akkor azt eltávolítom a vizsgálandók közül.

Alább látható az így kapott megoldásom.

**bemenet**: *(N,m)* Petri-háló, *T* a tranzakciók halmaza

**kimenet**: stubborn set

**for** minden **do**

|

| **for** minden és minden **do**

| | **if** **then**

| | ;

| | **end**

| **end**

| **if** **then**

| |

| |

| **end**

**end**

**while** amire F1 vagy F2 nem teljesül **do**

**end**

4. kódrészlet: Statikus algoritmus 3

# Eredmények kiértékelése

Annak érdekében, hogy be tudjam mutatni az elkészített algoritmusok hatékonyságát, méréseket végeztem több különböző Petri-hálón is. Erre a célra a PetriDotNethez kiadott modelleket használtam fel, ezek közül nem mindegyiken futott le minden algoritmus sikeresen. Ezen hálók jelentős része a Model Checking Contest[9] elnevezésű versenyen kiadott feladatok közül lett kiválasztva.

A méréseket a saját számítógépemen végeztem, mely az alábbi konfigurációt jelenti:

* Operációs rendszer: Windows 10 pro 64 bit, 2004 verzió
* Processzor: Intel Core i5-6500 3.20GHz
* Memória: HyperX 4GB FURY DDR4 2133MHz CL14 \*4 db (16GB)
* Alaplap: MSI Z170A TOMAHAWK
* SSD: SAMSUNG 120GB 650 SATA 3

## 8 bástya probléma

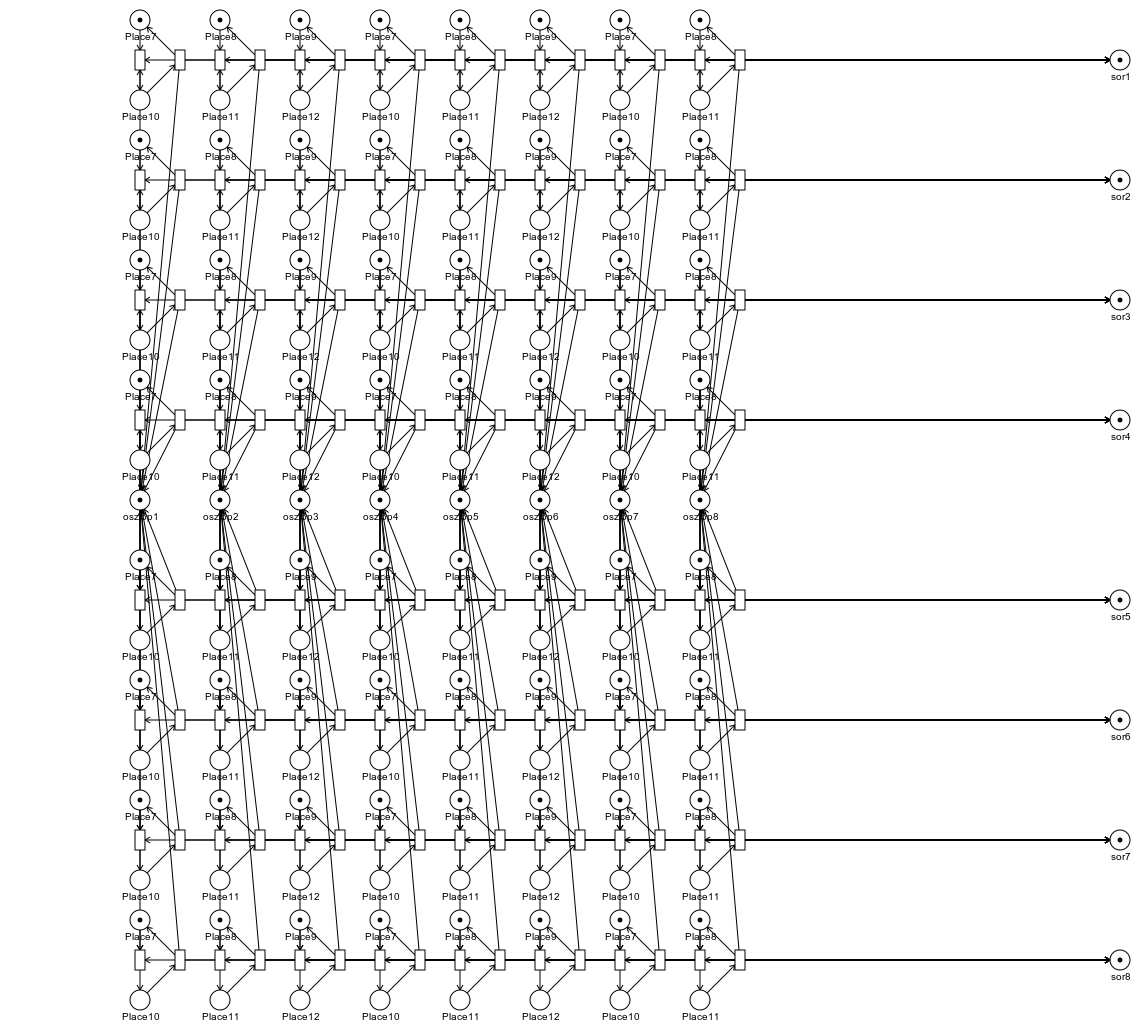
Az első mérésemet a 8 bástya problémát modellező Petri-hálón végeztem. A probléma azt vizsgálja, hogyan lehet sakktáblára lehelyezni úgy bástyákat, hogy ne üssék egymást. A 16. ábra mutatja a mérésekhez használt Petri-hálót, az 15. ábra pedig a mérési eredményeket. Az első számokat tartalmazó oszlop minden mérés esetében a teljes állapottér gráfra vonatkozik, a második a 3.2.2.1 bemutatott, a harmadik a 3.2.2.2 fejezetben bemutatott, az utolsó pedig a 0 fejezetben bemutatott módszerhez tartozó mérési eredményeket tartalmazzák. A későbbiekben ezekre sorra úgy fogok hivatkozni, hogy teljes állapottér generálása, első algoritmus, második algoritmus és harmadik algoritmus.

1. táblázat: A 16. ábra által mutatott Petri-hálóhoz tartozó mérési adatok

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 8rook | 8rookslow | 8rookss | 8rooksimp |
| állapotok száma | 1441729 | 65 | 65 | 65 |
| eltelt idő (ms) | 60829,265 | 117,9259 | 88,6202 | 53,1207 |

15. ábra: A 1.táblázat adatait mutatja vizuálisan

A teljes állapottér gráf majdnem másfél millió csúcsot tartalmaz, ebben az esetben hatékonynak bizonyult mind a három, a gráf méretének csökkentését szolgáló algoritmus és egységesen 65 csúcsot tartalmaznak, ami az eredeti érték elhanyagolható része. Futási idők esetén is jól látható a különbség. A teljes állapottér generálása egy egész percet igénybe vett, míg a másik három algoritmus kevesebb, mint 1 másodperc alatt lefutott. Ebben az esetben a leghatékonyabbnak az általam fejlesztett algoritmus bizonyult, futási ideje alig több, mint feleannyi volt, mint a nála eggyel lassabbnak.



16. ábra: A nyolc bástya problémát bemutató Petri-háló

## DPhil

Ez a Petri-háló az étkező filozófusok probléma[9] dinamikus változatát modellezi, mely annyit jelent, hogy a filozófusok leülhetnek és ott is hagyhatják az asztalt. A Dphil-10 nevű modellt használtam, a nagyobb számot tartalmazók már nem adtak belátható időn belül eredményt a rendelkezésemre álló memóriamennyiség mellett.

2. táblázat: A DPhil-10 Petri-hálóhoz tartalmazó mérési eredmények

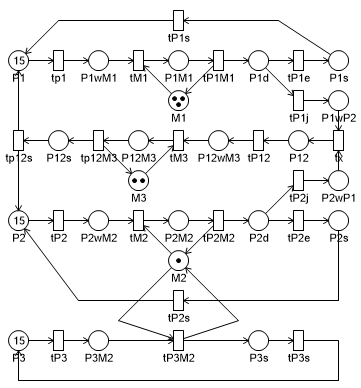
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | dphil10 | dphil10slow | dphil10ss | dphil10imp |
| állapotok száma | 1860498 | 38135 | 1542687 | 38135 |
| eltelt idő (ms) | 25914,1859 | 1310,1494 | 124089,996 | 746,8324 |

17. ábra A 2. táblázat értékeinek vizuális reprezentációja

Erre a modellre már láthatóan kevéssé ad hatékony megoldást a második algoritmus. Állapotok száma terén a teljes állapottér állapotszáma nem különbözik lényeges mértékben a második algoritmus által adott eredménytől, kb. 83%. A másik két stubborn set módszer viszont felderített állapotok terén azonos hatékonyságúnak bizonyult, az eredeti állapotok kb. 2%-át járták csak be. A teljes állapottér generálása majdnem 26 másodpercet vett igénybe, ehhez képest a második algoritmus futási ideje több, mint két perc volt, ami ugyancsak azt szemlélteti, hogy mennyire nem hatékony a működése. Ebben az esetben is a harmadik algoritmus bizonyult a leghatékonyabbnak.

## FMS-15

Ez a modell a Flexible Manufacturing System[9] nevet viseli, a 18. ábra mutatja be, hogy is néz ki. A 15-ös szám a P1, P2 és P3 helyhez tartozó tokenek számát jelöli. Ez a legkisebb méretű modell, amit tartalmaz a PetriDotNet, a nagyobbak esetében már nem sikerült értékelhető eredményt produkálnom a mérési eszközömön.



18. ábra: Az FMS-15 Petri-háló

3. táblázat: Az FMS-15 modellhez tartozó mérési adatok

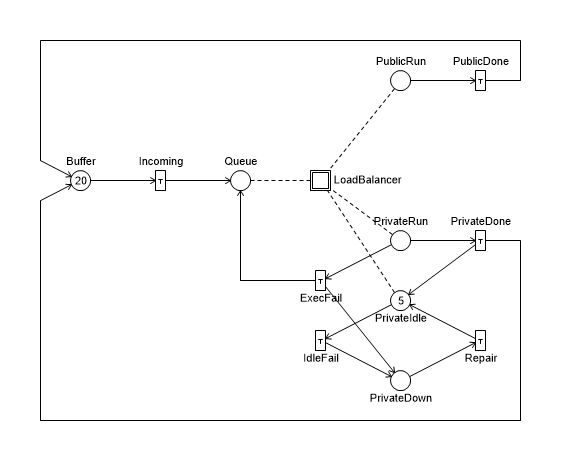
|  |  |  |
| --- | --- | --- |
|  | fms15slow | fms15imp |
| állapotok száma | 7196 | 4321 |
| eltelt idő (ms) | 85,8467 | 25,6091 |

19. ábra A 3. táblázatban szereplő adatok vizuális megjelenítése

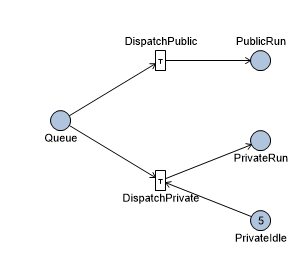
A 3. táblázat azért nem tartalmaz csak két számadatokkal rendelkező oszlopot, mert az első és a harmadik algoritmust sikerült csak lefuttatni a mérések során, a másik kettő túl nagy memória igényű volt (több, mint 24 GB), illetve a futási idő is nagyon hosszúra nyúlt (több óra). Ezzel szemben az első és a harmadik algoritmus futási ideje is egy másodperc alatt van. Ezen háló esetén is elmondható, hogy a harmadik algoritmus a leghatékonyabb.

## Hybrid-Cloud

A Hybrid-Cloud nevű Petri-háló (20. ábra) rendelkezik egy alhálóval (21. ábra) is és feladatok elosztását végzi több kiszolgáló között. A tranzíciókat jelző csomópontokon található egy T betű, ez azt jelenti, hogy időzítéseket is használ a háló arra, melyik tranzíció tüzeljen.



20. ábra: A hybrid-cloud nevű Petri háló



21. ábra: A hybrid-cloud nevű Petri-háló alhálója

4. táblázat: A hybrid-cloud modelhez tartozó mérési eredmények

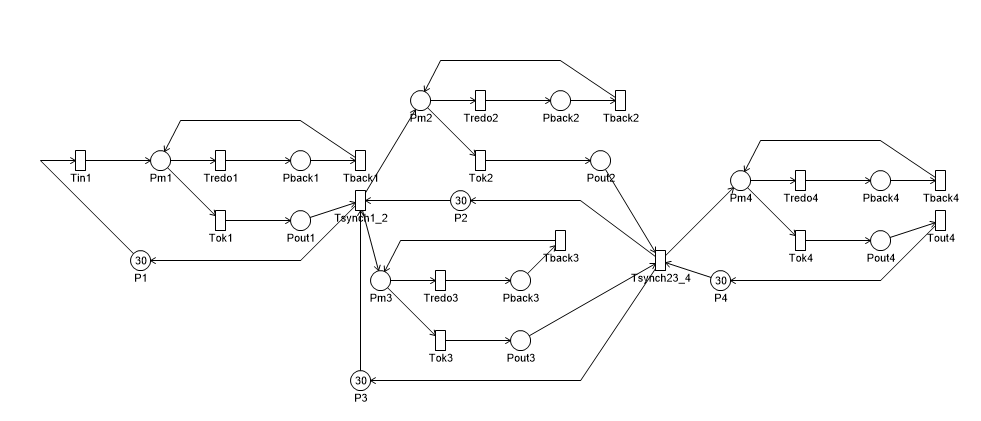
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | hybridcloudfull | hybridcloudslow | hybridcloudss | hybridcloudimp |
| állapotok száma | 126 | 22 | 26 | 22 |
| eltelt idő (ms) | 0,1225 | 0,0492 | 0,0606 | 0,0374 |

22. ábra A 4. táblázatban szereplő eredmények vizuális megjelenítése

A kiindulási modell viszonylag kevés helyet, tranzíciót és tokent tartalmaz, így a kapott mérési adatok is jelentősen kisebbek, mint a korábban bemutatott esetekben. Mindösszesen 126 állapot tartozik a teljes állapottérgráfhoz. A második algoritmus az összes bejárható állapotok 20,6%-át vizsgálja, míg az első és harmadik csak 17,5%-ot. A teljes állapottér vizsgálta is kevesebb, mint 1ms-t vesz igénybe, míg a futási idő szempontjából leghatékonyabb algoritmus ennek is csak körülbelül negyede.

## Kanban

Ez a háló a SMART benchmark eszközből lett kinyerve, egy Kanban rendszert[12] modellez. A 23. ábra mutatja az általam vizsgált modellt, csak azzal próbálkoztam, melyben a P1, P2, P3, P4 helyek 30 tokent tartalmaznak, mert a nagyobbak esetén már nem bizonyult elég erősnek a mérőeszközöm eredmények előállításához.



23. ábra: A Kanban-30 Petri-háló modellje

5. táblázat A Kanban-30 Petri-hálóhoz tartozó mérési eredmények

|  |  |  |
| --- | --- | --- |
|  | kanban-30slow | kanban-30imp |
| állapotok száma | 11778 | 610 |
| eltelt idő (ms) | 1451088 | 18922 |

24. ábra: A 5. táblázat által tartalmazott mérési eredmények vizuális megjelenítése

A 5. táblázat adataiból látszik, hogy ennél a modellnél se sikerült mind a négy algoritmust lefuttatnom, csak az első és a harmadik algoritmus adott eredményt. Ebben az esetben is próbálkoztam természetesen a másik kettő futtatásával is, ezek több óra és 23 GB memória használata után omlottak össze. A kapott eredmények között láthatóan nagy különbség van, a harmadik algoritmus alig több, mint 5%-át járta be az első algoritmus által vizsgált állapotoknak és időben pedig mindössze 1,3%-kát igényelte, mint a másik algoritmus futása.

## SlottedRing

A PetriDotNet által tartalmazott különböző méretű SlottedRing modellek közül a legkisebbet (SlottedRing5) tudtam csak vizsgálni, a nagyobb esetek nem adtak eredményt. A modell bővebb leírása megtalálható a [13] forrásban.

6. táblázat: A SlottedRing5 Petri-hálóhoz tartozó mérési eredmények

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | slottedRing5full | slottedring5slow | slottedring5ss | slottedring5imp |
| állapotok száma | 53856 | 19254 | 53856 | 51648 |
| eltelt idő (ms) | 413,5466 | 326,1819 | 5516,8258 | 2252,5164 |

25. ábra A 6. táblázat által tartalmazott mérési adatok vizuálisan megjelenítve

A SlottedRing5 volt az első olyan általam bemutatott modell, melyben nem sikerült csökkenést elérni a felfedezett állapotok számában valamelyik algoritmus segítségével. A második algoritmus pontosan ugyanannyi csúcsot vizsgál meg, mint a teljes állapottér és a harmadik algoritmus is bejárja a csúcsok kb. 96%-át. Az első algoritmus ennél kedvezőbb eredményt ad, itt a csúcsok kb. 36%-a kerül megvizsgálásra. Ahogy az állapotok számából is várható, a második és harmadik algoritmus futási ideje is hosszabb, mint a teljes állapottér bejárás, ugyanis ezek több számítási műveletet tartalmaznak. A második algoritmus több, mint tízszer annyi időt igényel, mint az állapottér felderítés, de a harmadik is több, mint ötszörösét. Ebben az esetben nem az én javításaimat is tartalmazó algoritmus bizonyult a leghatékonyabbnak, hanem az, amelyik elméletileg kevésbé hatékony stubborn seteket hoz létre.

## RoundRobin

A round robin egy hálózati terheléselosztásban használt algortimus, melynek lényege, hogy körben haladva minden résztvevő egyenlő időrést kap. Ez azt is jelenti, hogy nincs előnyben részesített résztvevő, mindenki azonos prioritással rendelkezik. Ez a modell is több méretben van megvalósítva a PetriDotNethez, én a legkisebbet tudtam csak vizsgálni.

7. táblázat A RoundRobin5 Petri-hálóhoz tartozó mérési eredmények

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | roundrobin10full | roundrobin10slow | roundrobin10ss | roundrobin10imp |
| állapotok száma | 23040 | 12672 | 23040 | 22784 |
| eltelt idő (ms) | 271,5359 | 665,6247 | 9435,2935 | 3840,5703 |

26. ábra: A 7. táblázat adatainak vizuális megjelenítése

A méréseim alapján megvizsgált állapotok tekintetében csak az első algoritmus tudott jelentősebb javulást felmutatni, ez a teljes állapottérgráf csúcsinak 55%-át érintette. A második algoritmus nem csökkentett semmit, a harmadik pedig csak elhanyagolható mértékben. A RoundRobin5 modell esetében az a helyzet állt elő, hogy a teljes állapottér gráf generálása vette igénybe a legkevesebb időt, az első algoritmusnak több, mint kétszer annyi kellett, a másodiknak majdnem 35-ször annyi, a harmadiknak pedig 14-szer annyi. Ennek az oka feltehetően az, hogy a stubborn setek keresése sok plusz számítási igényt jelent és hasonló mennyiségű csúcsra kellett ezeket elvégezni, mint a teljes állapottér vizsgálata esetében.

## Phil

A Phil modell a híres étkező filozófusok modell megvalósítása Petri-háló segítségével. A modell bővebb leírása megtalálható a [14] forrásban. A PetriDotNet ehhez a problémához is több méretű modellt is tartalmaz, sikeresen vizsgáltam a 10 és 100 méretű modelleket, azonban az 1000 már nem tudtam. Sajnos ebben az esetben sem áll rendelkezésre a Petri-háló modell vizuális formában.

8. táblázat: A Phil10 Petri-hálóhoz tartozó mérési adatok

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Phil10full | phil10 slow | Phil10ss | phil10imp |
| állapotok száma | 123 | 11 | 11 | 11 |
| eltelt idő (ms) | 0,4155 | 0,2849 | 0,2103 | 0,1429 |

9. táblázat: A Phil100 Petri-hálóhoz tartozó mérési adatok

|  |  |  |  |
| --- | --- | --- | --- |
|  | phil100slow | phil100ss | phil100imp |
| állapotok száma | 101 | 101 | 101 |
| eltelt idő (ms) | 1407,2626 | 302,3803 | 455,0177 |

27. ábra A 8. táblázat adatinak vizuális megjelenítése

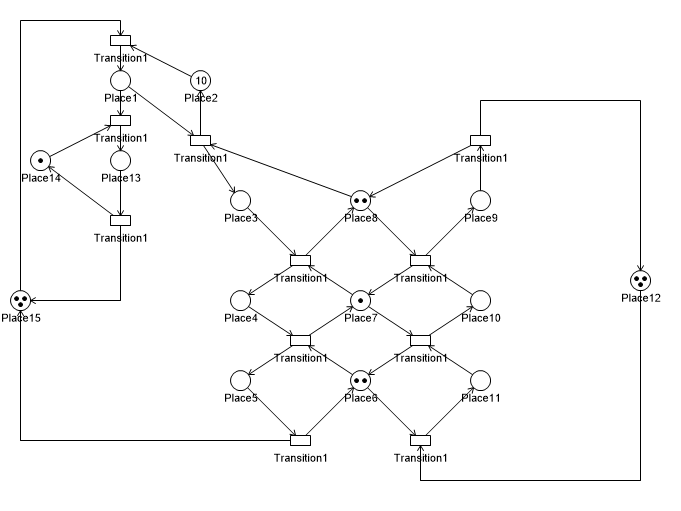
28. ábra: A 9. táblázat időadatinak megjelenítése

A Phil10 esetében mind a három algoritmus jelentős javulást eredményez a teljes állapotgráf bejárásához képest, mindössze az összes állapot 9%-át érintik. Időben is kedvezőbb eredményt adnak a stubborn set algoritmusok, a harmadik algoritmus a leghatékonyabb, ez időben harmada, mintha az összes lehetséges állapotot vizsgálnánk.

A Phil100 esetén nincs a mérési eredmények között a teljes állapottérgráf vizsgálata. Ennek az az oka, hogy a saját számítógépemen ezt a mérést nem sikerült elvégeznem, ezért próbálkoztam egy másikon is, ami 32 GB RAM-ot és egy Intel Core i9 processzort tartalmaz. Ezen két és fél óra után leállítottam a mérést, mert nem hozott eredményt, de a rendelkezésre álló összes memóriát már sokkal hamarabb megtöltötte ennél, így a lapozófájlok használata miatt tovább nőtt a várható futási idő. Sikerült hozzáférést szereznem egy Intel Xeon processzorral és 100 GB RAM-mal rendelkező szerver számítógéphez, ezen sem sikerült eredményt kapnom a teljes állapottér esetében több órányi számítás és az összes memória kitöltése után sem. Nem készítettem diagrammot a felfedezett állapotok számához, mert minhárom vizsgált algoritmus egységesen 101 állapotot járt be. Az algoritmusok futásához szükséges idő itt sem volt nagy, a leglassabb esetben is mindösszesen másfél másodperc körül. Ebben az esetben érdekes eredmény, hogy a harmadik algoritmus rosszabbul teljesít felhasznált idő szempontjából, mint a második, de még így is mindkettő hatékonyabb, mint az első algoritmus.

## Dfms

A Dfms Petri-háló modell volt az egyetlen, amelyiknek mind a PetriDotNet által tartalmazott méretű modelljére sikerült elvégeznem a méréseket. Ez a Dfms-10, Dfms-100, Dfms-1000, Dfms-10000. A különbség köztük mindössze annyi, hogy a Place2 elnevezésű hely mennyi tokent tartalmaz.



29. ábra: A Dfms10 Petri-háló modellje

10. táblázat: Dfms10 Petri-hálóhoz tartozó mérési eredmények

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | dfms-10full | dfms-10slow | dfms-10s | dfms-10imp |
| állapotok száma | 3238 | 1222 | 3222 | 1106 |
| eltelt idő (ms) | 6,4857 | 6,6696 | 27,6637 | 3,4927 |

11. táblázat: Dfms100 Petri-hálóhoz tartozó mérési eredmények

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | dfms-100full | dfms-100slow | dfms-100s | dfms-100imp |
| állapotok száma | 33028 | 12112 | 33012 | 11096 |
| eltelt idő (ms) | 81,6323 | 71,655 | 291,7853 | 38,4564 |

12. táblázat: Dfms1000 Petri-hálóhoz tartozó mérési eredmények

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | dfms-1000 | dfms-1000 slow | dfms-1000ss | dfms-1000imp |
| állapotok száma | 330928 | 121012 | 330912 | 110996 |
| eltelt idő (ms) | 1051,915 | 636,5637 | 3042,177 | 532,4739 |

13. táblázat: Dfms10000 Petri-hálóhoz tartozó mérési eredmények

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | dfms-10000 | dfms-10000 slow | dfms-10000ss | dfms-10000imp |
| állapotok száma | 3309928 | 1210012 | 3309912 | 1109996 |
| eltelt idő (ms) | 10316,9342 | 6282,61 | 29959,26 | 5100,162 |

30. ábra A 10. táblázat adatainak vizuális megjelenítése

Csak a legkisebb Petri-hálóhoz tartozó mérésekhez készítettem grafikonokat, ennek oka, hogy a táblázatok is jól mutatják: arányaiban nem lesz eltérés a különböző algoritmusok futása között. A felfedezett állapotok szempontjából az első és a harmadik algoritmus a hatékony ezekben az esetekben, a harmadik egy kicsit hatékonyabb. Szükséges idő szempontjából az első algoritmus csak kisebb javulást eredményez, a második az eredetileg szükséges idő sokszorosát használja fel (itt is kikövetkeztethető az állapotok számából és a többletműveletekből), a harmadik átlagosan pedig fele annyit igényel, mint a teljes állapottér generálása.

31. ábra: A teljes állapottér gráfhoz tartozó állapotok száma és a felderítéshez szükséges idő

A 31. ábra bemutatja, hogy a tokenszám növelésével hogyan változik az állapotok és a szükséges idő. Összességében elmondható, hogy a 10-szeres szorzó megfigyelhető mindkét adat esetében.

## Algoritmusok értékelése

Általánosságban elmondható, hogy az általam másodiknak kipróbált stubborn set algoritmust nem érdemes használni bármelyik másikkal szemben, mert több esetben sem eredményezett semmilyen javulást az állapotok számában, de a futási ideje magasabb volt. Ahol pedig volt javulás, ott is jellemzően alulmaradt a másik két megoldással szemben.

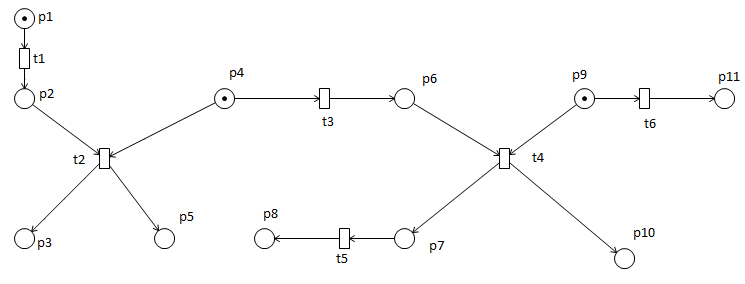
Az első és a harmadik algoritmus közül viszont már nem egyértelműen lehet kiválasztani a jobb megoldást. Az első összességében talán stabilabban hoz javulásokat, de a harmadik is az esetek többségében jelentős javulást eredményez és ilyenkor rendszerint mind időben mind állapotok számában megveri az elsőt.

# Működés bemutatása példákon

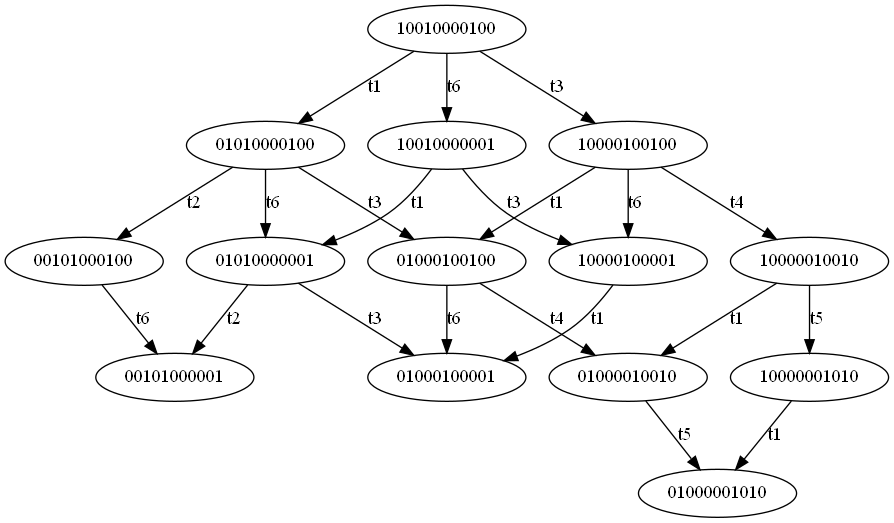
Ebben a fejezetben néhány kisebb példán keresztül bemutatom, hogyan működnek az általam megvalósított algoritmusok. Ehhez az elkészített dll-ben eltároltam, mely állapotok melyik másik állapotokból érhetőek el, majd ezt elmentettem szöveges fájlba, olyan formátumban, hogy a Graphviz segítségével feldolgozható legyen. A Graphviz feladat volt, hogy a szöveges eredményeket átalakítsa képpé.

Az első bemutatott Petri-háló már szerepelt korábban a dolgozatomban (12. ábra), ezen keresztül mutattam be a dinamikus stubborn set működését, így már szerepelt a hozzá tartozó teljes állapottérgráf (13. ábra) és a dinamikus stubborn set módszert alkalmazva kapott állapottér gráf is (14 ábra). Erre a hálóra a bemutatott három algoritmus egyike se tudott javulást eredményezni a meglátogatott állapotok számában, ezért nem teszek be képet se az eredményül kapott gráfról sem.

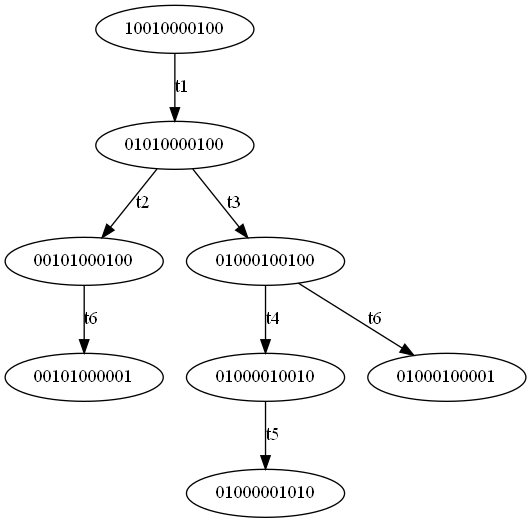
A következő példaként szolgáló Petri-hálót[4] a 32. ábra mutatja. A 33. ábra mutatja a hozzá tartozó teljes állapotér gráfot, ez 14 darab csúcsból áll, a dinamikus stubborn set módszert használva kapott eredmény (34. ábra) pedig 8 csúcsot tartalmaz.



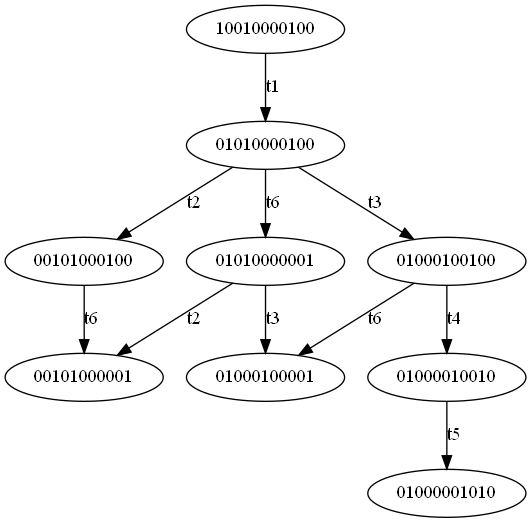
32. ábra: Példaként bemutatott Petri-háló



33. ábra: A 32. ábra által mutatott Petri-háló teljes állapottérgráfja



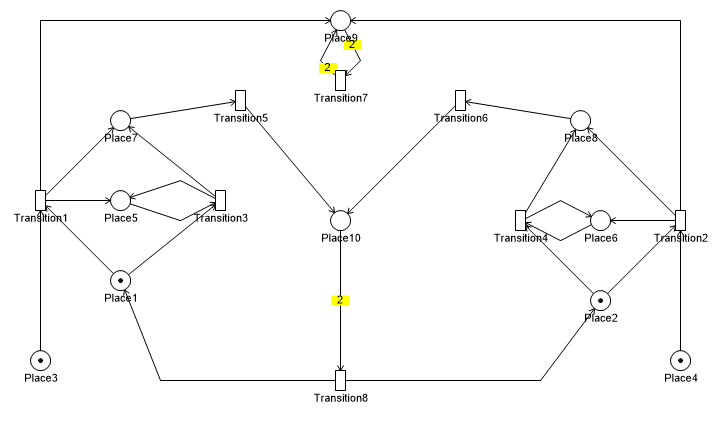
34. ábra: A 32. ábra Petri-hálójának dinamus stubborn set módszerrel csökkentett részleges állapottér gráfja



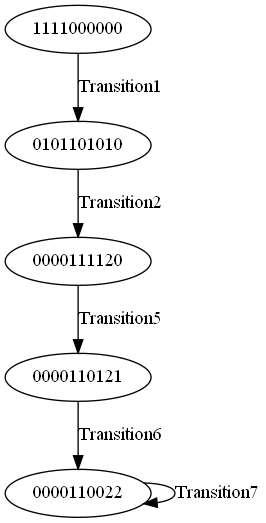
35. ábra: A 32. ábra Petri-hálójához tartozó statikus stubborn set módszerekkel kapott részleges állapottér gráfja

Mind a három statikus módszert alkalmazó algoritmus azonos eredményt adott, ezt mutatja a 35. ábra. Az így kapott gráfnak 9 darab csúcsa van, ez is lényeges javulást jelent az eredetihez képest. A különbség az a dinamikus módszer adta eredménytől, hogy a 01010000001 csúcs is szerepel a gráfban.

A harmadik példa szerepel a [1] cikkben is, a Petri-háló (36. ábra) 8 tranzícióból és 10 helyből áll, és több ciklust is tartalmaz.



36. ábra: Példa Petri-háló



37. ábra: A 36. ábra által mutatott Petri-hálóhoz tartozó részleges állapottérgráf

A teljes állapottérgráfot a 38. ábra szemlélteti (függelékben található), 14 állapotból áll. A 3.2.2.2 bekezdésben bemutatott algoritmus a 39. ábra (függelékben található) által mutatott eredményt adja. Itt az eredetihez képest nem csökken a bejárt állapotok száma, csak a tüzelendő tranzícióké. A Transition7 9 állapotból is tüzelhet és minden esetben hurokélként viselkedik, az algoritmus ezt csökkentette le, hogy csak egy állapotból tüzeljen. A 3.2.2.1 fejezetben és a 0 fejezetben leírt algoritmus is a 37. ábra által mutatott eredményt adja, ez a gráf csupán 5 csúcsot és 5 tranzíciót tartalmaz. A Transition7 ebben a gráfban is szerepel egyszer hurokélként. Az összesen 5 bejárt állapot jelentős javulást jelent az eredeti 14-hez képest, ez a gráf a hurokélen felül már nem tartalmaz ciklust se.

# Összefoglaló

A diplomamunkám elkészítése során irodalomkutatást végeztem a részleges rendezést használó algoritmusok között, annak érdekében, hogy azokat Petri-hálókra alkalmazva csökkenteni tudjam a teljes állapottér gráfjukat. A stubborn set módszerre esett a választásom, ezt vizsgáltam meg részletesebben is.

Megismerkedtem a PetriDotNet elnevezésű alkalmazással, ezt használtam arra, hogy Petri-háló modelleken alkalmazni tudjam a kiválasztott módszereket. Megtanultam, hogyan kell plug-int készíteni PetriDotNet-hez és milyen különböző adatszerkezeteket használ.

A szakirodalomban való keresgélés során megismerkedtem a stubborn setek dinamikus definíciójával. A dinamikus stubborn set olyan adatokat is felhasznál a teljes állapottér gráf csökkentésére, melyek nem állnak rendelkezésre csak a teljes gráf ismeretében. Ez nem megfelelő megoldás, mert ezek mérete exponenciálisan is nőhet a kiindulási modell növekedésével, pont ezért szeretnénk csökkenteni azon. A különböző statikus stubborn set módszerek lényege, hogy egy használható algoritmust nyújtanak, mely kielégíti a dinamikus definíciót is. A diplomaterv készítése során három ilyen algoritmust valósítottam meg az elkészült plug-inben, illetve kiindulásként és összehasonlítási alapként a teljes állapottér gráf bejárására is nyújtottam megoldást.

Az elkészült algoritmusokat az alapján vizsgáltam, hogy hány különböző állapotot érintenek és mennyi időt vesz igénybe a lefutásuk, a mérések elvégzéséhez a PetriDotNet által tartalmazott Petri-hálókat használtam fel. Az eredmények azt mutatták, hogy két algoritmus közül nem lehetett eldönteni, melyiket érdemes használni, mert általában az egyik hatékonyabban teljesített, néhány esetben viszont nagyon alulmaradt a másikkal szemben.

Továbbfejlesztési lehetőség lehet egy becslő eszköz készítése, ami a bemeneti modellből a helyek, tranzíciók és tokenek számából becslést ad az állapottér gráf várható méretére. Ez segíthetne abban, hogy egyáltalán érdemes-e azt vizsgálni vagy biztosan nem rendelkezünk a kellő erőforrásokkal.

További fejlesztési lehetőség más részleges rendezést használó algoritmusok használata és azok összehasonlítása a stubborn set módszer által adott eredményekkel.

# Irodalomjegyzék

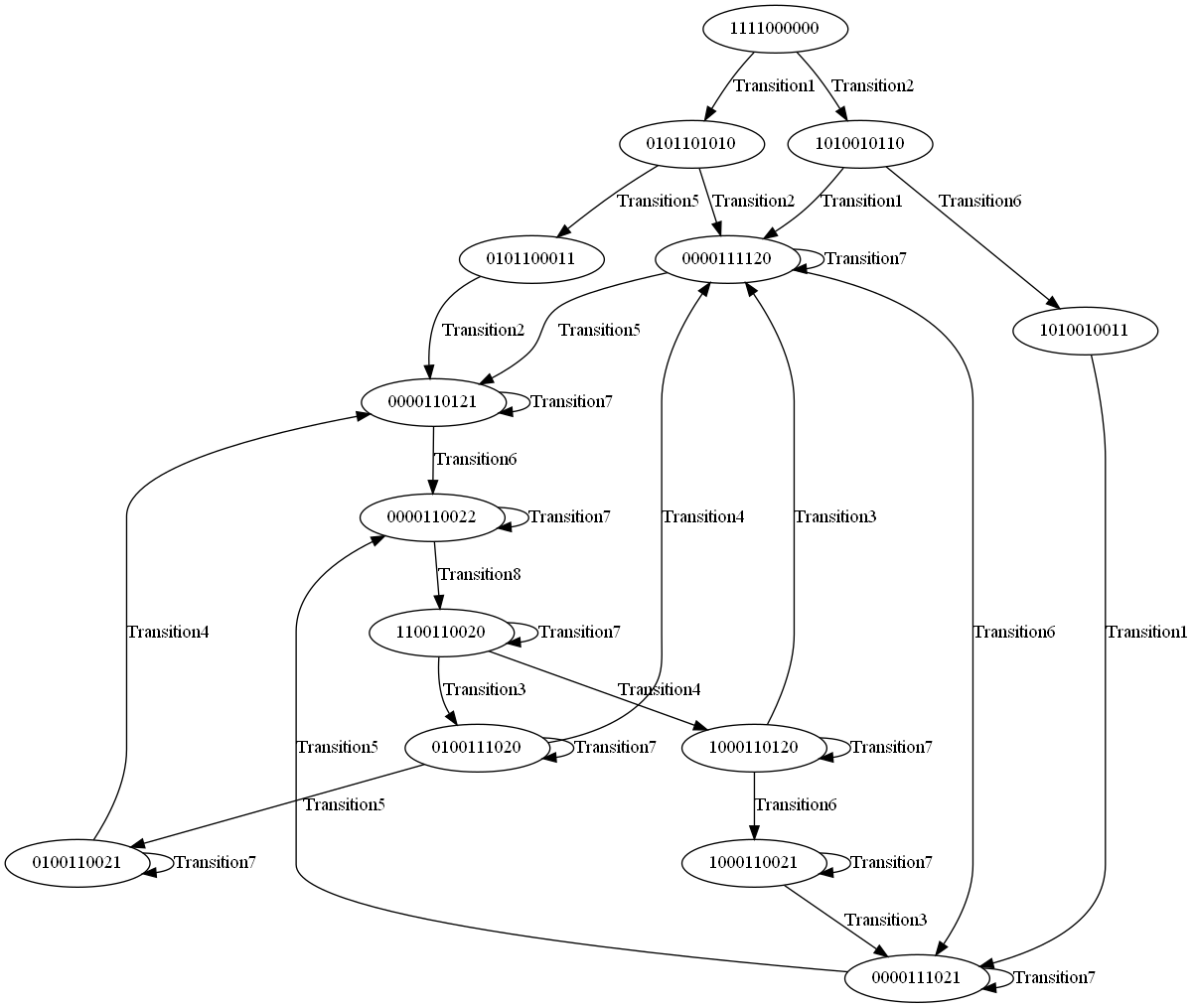
1. Kristensen, L.M., Schmidt, K. & Valmari, A. Question-guided stubborn set methods for state properties. Form Method Syst Des 29, 215–251 (2006). <https://doi.org/10.1007/s10703-006-0006-1>
2. Hajdu Ákos, Mártonka Zoltán. Diszkrét dinamikus rendszerek viselkedésének felderítése ellenpélda-alapú absztrakció finomítás (CEGAR) segítségével. TDK-dolgozat. 2012., online: <https://inf.mit.bme.hu/sites/default/files/edu/report/2012.%20%C5%91sz/documentation/DPUZIA-Diszkr%C3%A9t%20dinamikus%20rendszerek%20viselked%C3%A9s%C3%A9nek%20felder%C3%ADt%C3%A9se%20ellenp%C3%A9lda-alap%C3%BA%20absztrakci%C3%B3%20finom%C3%ADt%C3%A1s%20(CEGAR)%20seg%C3%ADts%C3%A9g%C3%A9vel.pdf>
3. Antti Valmari and Henri Hansen. Can Stubborn Sets Be Optimal?. Tampere University of Technology, Department of Software Systems PO Box 553, FI-33101 Tampere, Finland online: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.227.8112&rep=rep1&type=pdf>
4. Antti Valmari and Henri Hansen. Stubborn Set Intuition Explained. Department of Mathematics, Tampere University of Technology P.O. Box 553, FI–33101 Tampere, Finland. online: <http://ceur-ws.org/Vol-1591/paper15.pdf>
5. A petridotnet keretrendszer honlapja. (utoljára megtekintve: 2020.12.10.) <http://petridotnet.inf.mit.bme.hu/>
6. Pataricza András, editor. Formális módszerek az informatikában. Typotex, 2. kiadás, 2005.
7. A Graphviz honlapja (utoljára megtekintve: 2020.12.10) <https://graphviz.org/>
8. Valmari, A. (1998). The state explosion problem. Lecture Notes in Computer Science, 429–528. doi:10.1007/3-540-65306-6\_21
9. A Model Checking Contest honlapja: <https://mcc.lip6.fr/index.php> (utoljára megtekintve: 2020.05.10.)
10. https://mcc.lip6.fr/2020/pdf/PhilosophersDyn-form.pdf (utoljára megtekintve: 2020.05.10.)
11. <https://mcc.lip6.fr/2020/pdf/FMS-form.pdf> (utoljára megtekintve: 2020.05.10.)

1. <https://mcc.lip6.fr/2020/pdf/Kanban-form.pdf> (utoljára megtekintve: 2020.05.10.)

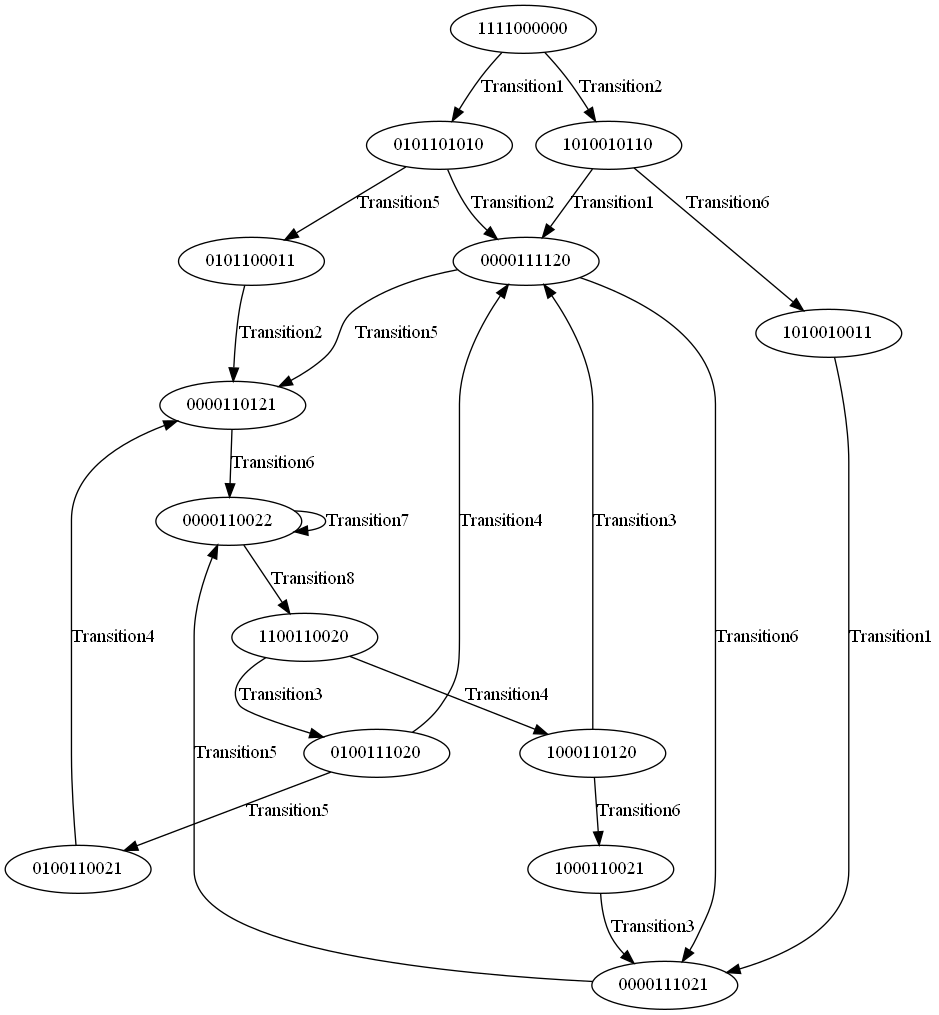
1. <https://mcc.lip6.fr/2020/pdf/TokenRing-form.pdf> (utoljára megtekintve: 2020.05.10.)

1. <https://mcc.lip6.fr/2020/pdf/Philosophers-form.pdf> (utoljára megtekintve: 2020.05.10.)

# Függelék



38. ábra: A 36. ábra által mutatott Petri-hálóhoz tartozó teljes állapottérgráf



39. ábra: A 36. ábra által mutatott Petri-hálóhoz tartozó részleges állapottérgráf