KOOPERÁCIÓ ÉS GÉPI TANULÁS LABORATÓRIUM

Tervkészítés

Elméleti segédlet

Készítette:
Kovács Dániel László
(dkovacs@mit.bme.hu)

Méréstechnika és Információs Rendszerek Tanszék
Budapesti Műszaki és Gazdaságtudományi Egyetem

2012, október.

Tartalomjegyzék

I. Bevezetés

Jelen segédlet a Budapest Műszaki és Gazdaságtudományi Egyetem Méréstechnika és Információs Rendszerek tanszéke által szervezett „*Intelligens Rendszerek*” M.Sc. szakirány „*Kooperáció és gépi tanulás (VIMIM223)*” c. laboratóriumának „*Tervkészítés*” c. laborgyakorlatához készült. Célja a hallgatók gyakorlatra való elméleti és gyakorlati felkészítése, továbbá a tervkészítés témakörének általános bemutatása.

A ***tervkészítés*** nagyjából az 1960-as évek óta tekinthető aktív kutatási területnek. A kutatás célja olyan tervkészítő rendszerek kidolgozása, amelyek a valós adatok, tapasztalatok fényében képesek hatékony és megbízható cselekvéstervek előállítására. Példának okáért közismert, hogy 1991-ben az amerikai hadsereg tervkészítő rendszereket vetett be az Öböl-háború során harci egységei vezérlésére, ellátásuk ütemezésére, stb. Ezzel a technológiával igen nagy előnyre tettek szert, hiszen a tervkészítő rendszerek temérdek olyan lehetőséget, eshetőséget is képesek voltak mérlegelni, amit a másik fél hadi-vezetői, stratégái már nem tudtak számításba venni. Ezen felül – a beérkező információk alapján – szinte „azonnali” döntést, stratégia-módosítást, vagy éppen új stratégiát tudtak javasolni, miközben az ellenfélnél ugyanez emberi döntéshozást, tervezést igényelt, ami jóval lassabb.

Tehát a tervkészítési eszközök használatának köszönhetően az amerikai haderők (már a tényleges harc megkezdése előtt) nem csak minőség, hanem gyorsaság tekintetében is előnyre tettek szert (az egyéb technológiai előnyökről nem is beszélve). A modern haditechnika alkalmazása így válhatott teljessé. A függelékben (aminek ismerete nem szükséges a kapcsolódó laborgyakorlat teljesítéséhez) még más ehhez hasonló, többnyire békésebb alkalmazási példákat is felsorolunk, továbbá áttekintjük a tervkészítő rendszerek elméleti alapjait. Előtte azonban a segédlet célkitűzéséhez híven bevezetjük a laborgyakorlat elvégzéséhez szükséges főbb elméleti és gyakorlati alapismereteket. Ennek során egy viszonylag összetett tervkészítési mintapéldát is megoldunk.

Hogyan érdemes olvasni az anyagot? – merülhet föl a kérdés. A kapcsolódó laborgyakorlatra készülő hallgatóknak elegendő a II. fejezet, vagy esetleg azon belül is csak a II.2-es, II.3-as (azon belül II.3.1-3), és II.4-es alfejezet ismerete. Az érdeklődőbb Olvasó számára azonban javallott a függelék (lásd. IV. fejezet) elolvasása is, amin belül mélyebben is megismerkedhet a tervkészítés elméletével, és történetével. A II.2-es, és II.3-as alfejezet számos forráskód-részletet tartalmaz. Akinek kényelmesebb, esetleg első olvasatra át is ugorja ezeket, majd később, szükség szerint visszatérhet hozzájuk.

A segédlet elkészítéséhez nyújtott segítségéért kiemelt köszönet illeti Dr. Kovács András (akovacs@sztaki.hu), aki számos hasznos javaslattal, tapasztalattal, oktatási segédanyaggal, és gyakorlati eszközzel támogatta a kapcsolódó laborgyakorlat, s így e segédlet létrejöttét.

II. Tervkészítés labor

Ebben a fejezetben bevezetjük a tervkészítés alapvető elméleti és gyakorlati alapfogalmait, megoldunk egy tervkészítési mintapéldát, és bemutatjuk a laborgyakorlat szoftveres és hardveres infrastruktúráját.

II.1. Elméleti alapfogalmak

A tervkészítést végző autonóm rendszert tekintsük tervkészítési környezetbe ágyazott ***ágens****nek*. A tervkészítő ágensnek nem mindig elegendő pusztán csak az aktuális megfigyelést, vagy éppen saját aktuális benső állapotát figyelembe vennie ahhoz, hogy a megfelelő döntést meghozhassa (lásd. reaktív ágensek). A megfelelő cselekvés kiválasztása legtöbbször bizonyos szintű „előrelátást" kíván, aminek során az ágens különböző cselekvés-sorozatokat mérlegel annak érdekében, hogy kiválaszthassa közülük a céljainak legmegfelelőbbet. Ezt a folyamatot nevezik ***tervkészítés****nek.*

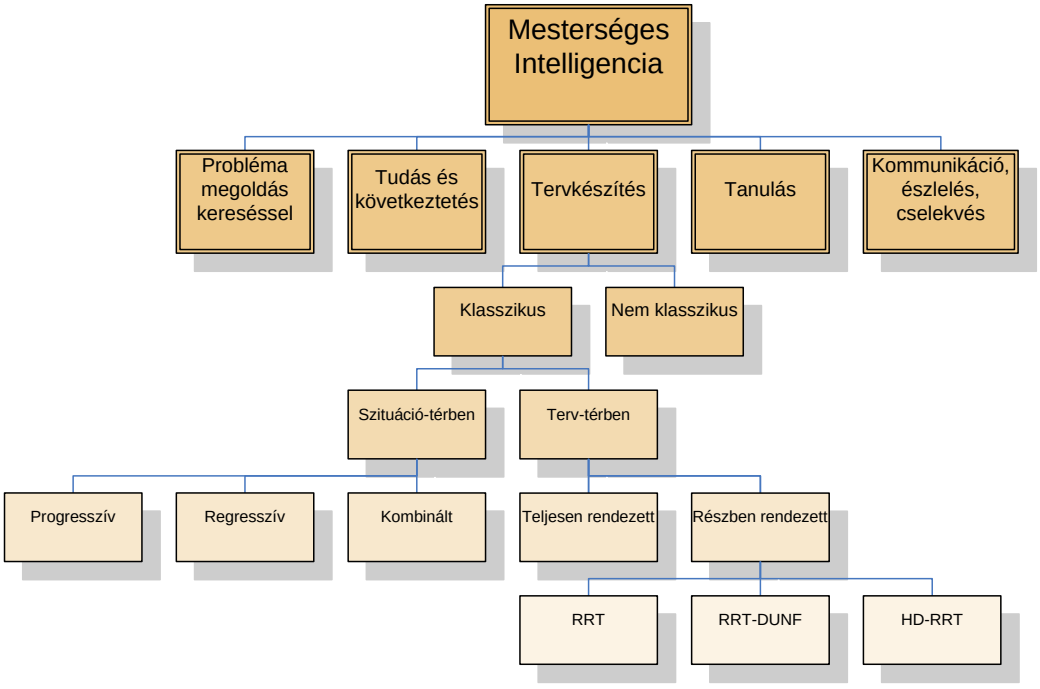
Probléma*nak* nevezik az ágens-környezet *kiinduló állapotának*, *cél-állapotainak* és lehetséges *cselekvéseinek* együttesét. Ekkor a ***probléma megoldása*** a cselekvések egy olyan sorozata, melyek végrehajtása a kiinduló állapotból a cél-állapotok valamelyikébe vezet. A cél-állapot megadása lehet explicit, vagy implicit. Utóbbi esetben többnyire valamiféle *cél-függvény* segítségével dönthetjük el, hogy az adott állapot része-e a cél-állapotok halmazának. Cél-állapotnak tekinthetjük például ekkor mindazon állapotokat, melyekhez a cél-függvény – a hozzájuk vezető cselekvés-sorozat ismeretében – egy-egy adott értéket rendel (lehet ez akár a függvény maximuma, minimuma, vagy bármely más értéke). A problémákat a következőképp osztályozhatjuk:

- *Egyállapotú problémák* azok, amelyek olyan teljesen hozzáférhető (kvázi determinisztikus) környezetet írnak le, ahol az egyes cselekvések kimenetele az ágens számára teljes egészében ismert.
- *Többállapotú problémák* azok, amelyek olyan, nem teljesen hozzáférhető (kvázi nem-determinisztikus) környezetet írnak le, ahol az egyes cselekvések lehetséges kimenetelei az ágens számára teljes egészében ismertek.
- *Eshetőségi problémák* azok, amelyek olyan, nem teljesen hozzáférhető (kvázi nem-determinisztikus) környezetet írnak le, ahol az egyes cselekvések lehetséges kimenetelei az ágens számára csak részben ismertek.
- *Felderíthetőségi problémák* azok, amelyek olyan, nem teljesen hozzáférhető (kvázi nem-determinisztikus) környezetet írnak le, ahol az egyes cselekvések lehetséges kimenetelei az ágens számára (kezdetben) egyáltalán nem ismertek.

A problémák definíciója jól láthatóan az ágens szemszögéből, a környezet viszonylatában történt. Ez nem jelent megkötést a problémák körére vonatkozólag, mivel az ágens, illetve a környezet kellő általánosítása esetén az ágens-környezet által reprezentált probléma ekvivalens lehet bármely általános értelemben vett problémával, amellyel egy autonóm probléma-megoldó rendszer szembesülhet. E problémák megoldására szolgálnak a tervek.

***Terv**nek* nevezzük *lépések* egy halmazát és a rajtuk értelmezett *kényszerek* és *relációk* összességét. A tervnek ez a – már-már megfoghatatlanul általános – definíciója nem véletlen, ugyanis a különböző tervkészítő módszerek más-más módon definiálják és reprezentálják mind a lépéseket, mind a kényszereket, mind pedig a relációkat, miközben gyakorlatilag mind a fenti, általános definíció speciális esetei.

Az előbb bevezetett fogalmak lényegében mind-mind a ***klasszikus*** (avagy más néven determinisztikus) ***tervkészítés*** témaköréből erednek. Ennek felépítése és elhelyezkedése a Mesterséges Intelligencia (MI) tárgyterületén belül a következő:



II-1. ábra: a Klasszikus tervkészítés, mint az MI egy részterülete

A II-1. ábra szerint az MI öt részre tagolható, amin belül a „Tervkészítés” egy egészen különálló fejezetet képez. Ezen belül beszélhetünk ún. „Klasszikus”, és „Nem klasszikus” tervkészítésről. Az előbbi csak azokkal az esetekkel foglalkozik, ahol a tervkészítő ágens környezete statikus (azaz csak az ágens idézhet elő változást), determinisztikus, és teljesen hozzáférhető, míg az utóbbi foglalkozik minden egyéb esettel (pl. nem teljesen hozzáférhető, dinamikus, sztochasztikus környezetekkel). Mivel a segédlethez kapcsolódó laborgyakorlat során kizárólag klasszikus tervkészítéssel kívánunk foglalkozni, ezért most ennek a résznek a felosztását vizsgáljuk tovább.

A klasszikus tervkészítés két további részre osztható: ***szituáció-térben***, és ***terv-térben történő keresés****re*. Az előbbi lényegében a „Probléma megoldás kereséssel” részben tárgyalt klasszikus kereső algoritmusokat fedi, ahol a keresési tér állapotai az ágens-környezet állapotainak felelnek meg, míg az operátorok az ágens cselekvései. Attól függően, hogy a kezdőállapotból a célállapot felé, vagy fordítva, esetleg mindkét irányban egyszerre haladunk, beszélhetünk ***progresszív***, ***regresszív***, és ***kombinált*** tervkészítésről (a szituáció-térben).

A terv-térben történő tervkészítés során a keresési tér állapotai tervek, míg az operátorok e terveket finomítják, módosítják. Ennek a szemléletváltásnak előnye, hogy jelentős mértékben csökkenti a tervkészítés komplexitását, mivel amíg a szituáció-térben az operátorok (azaz az állapot-változásért felelős ágens-cselekvések) száma tetszőlegesen nagy lehet, s így keresési tér elágazási tényezője is tetszőlegesen nagy, addig a terv-térben csak konstans számú operátor áll rendelkezésünkre, s így az elágazási tényező is konstans korlátos. Ez jelentős hatékonyságnövekedést eredményez általában.

A terv-térben kereshetünk ***teljesen***, és ***részben rendezett*** tervek közt. Az előbbi esetben a lépések sorrendje fix, míg az utóbbi esetben előfordulhatnak olyan lépések, melyek sorrendje nem kötött (és végül majd csak a terv végrehajtásakor dől el). Ennek több előnye is van (pl. a párhuzamos végrehajtás). A fentiekből kifolyólag a jelen segédlethez kapcsolódó laborgyakorlat során főként részben rendezett tervkészítéssel foglalkozunk.

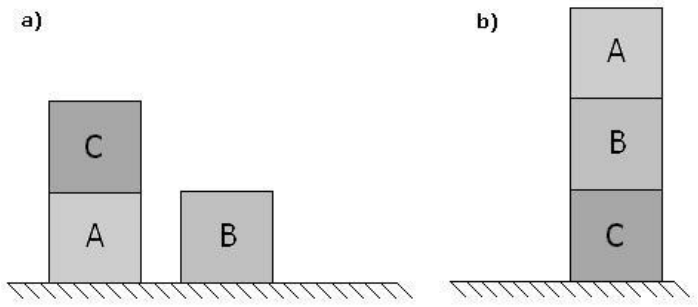
A részben rendezett tervkészítés három fő részre (algoritmusra, módszerre) tagolható: ***RRT*** (R*észben* R*endezett* T*ervkészítés*), ***RRT-DUNF*** (R*észben* R*endezett* T*ervkészítés* D*iszjunkcióval*, U*niverzális kvantorokkal*, N*egálással és* F*eltételes következményekkel*), és ***HD-RRT*** (H*ierarchikus* D*ekompozíciós* R*észben* R*endezett* T*ervkészítés*). Mindhárom algoritmus részletes leírása megtalálható a függelékben.

II.2. Gyakorlati alapfogalmak

Az elméleti bevezetőben láthattuk, hogy a jelen segédlethez kapcsolódó, Tervkészítés c. laborgyakorlat során klasszikus, azon belül is kizárólag részben rendezett tervkészítéssel foglalkozunk. Ebben a fejezetben tehát ennek gyakorlati megvalósítása kerül terítékre. Megismerkedünk a ***PDDL*** (P*lanning* D*omain* D*efinition* L*anguage*) nyelvvel (McDermott és társai, 1998), amely a de facto szabvány a klasszikus tervkészítési problémák leírására. E mellett megismerkedünk egy könnyen kezelhető, RRT-alapú, ***LPG*** (L*ocal search for* P*lanning* G*raphs*) nevezetű tervkészítő alkalmazással is (Gerevini és Serina, 2002)[1], amit – amolyan fekete dobozként – PDDL-ben leírt problémák megoldására fogunk használni. A PDDL lényegét leginkább egy példán keresztül érthetjük meg (lásd. II.2.1).

II.2.1. Sussman anomália

A probléma a következő: *adott egy asztal, és rajta kockák. Célunk, hogy a II-2. ábra (a) részén látható kiindulási állapotból az ábra (b) részén látható cél-állapotba jussunk pusztán csak a kockák rakosgatásával. A mozgatott kockán más kocka nem foglalhat helyet, továbbá – az asztalt leszámítva – csak olyan kockára tehetjük, amelyiken még nincsen kocka.*



II-2. ábra: Sussman anomália: (a) kezdőállapot, (b) célállapot

A probléma ***Sussman anomália*** (Sussman, 1973) néven vált ismertté, mivel egy időben (mikor a tervkészítés még igencsak gyerekcipőben járt) az akkori tervkészítő módszerek nemigen tudtak rá megoldást adni. Ennek oka, hogy a rész-célok („A van B-n” és „B van C-n”) összefüggnek egymással[2]. Természetesen a mai tervkészítő alkalmazások, melyek alapja a részben rendezett tervkészítés (RRT), már könnyedén megoldják az ilyen, és ehhez hasonló, akár jóval összetettebb problémákat. Próbáljuk meg tehát mi is – egy alkalmasan választott tervkészítő eszköz segítségével – megoldani e problémát!

[1] Az alkalmazás ingyenesen letölthető a következő URL-ről: http://zeus.ing.unibs.it/lpg

[2] Amellett, hogy az „*A-t B-re rakom*”, és a „*B-t C-re rakom*” cselekvések nem hajthatók végre egymástól függetlenül (pl. egyszerre), ráadásul még az is nehezíti a helyzetet, hogy C rajta van A-n, miközben A nincsen B-n, aminek C-n kellene lennie… Ez utóbbi „körkörösség” C asztalra helyezésével oldható fel.

II.2.2. PDDL és LPG

1. Első lépésben válasszunk egy alkalmas tervkészítő szoftvert, amely képes PDDL nyelven leírt problémák megoldására.
2. Reprezentáljuk a fenti problémát PDDL nyelven, és oldjuk meg a választott szoftver segítségével.

Mindez igen egyszerűnek tűnik, azonban a látszat csal. Ha van is alkalmas tervkészítő eszköz a birtokunkban, a probléma leírása még akkor is igen sok megfontolást és körültekintést igényel. Ráadásul a tervkészítő szoftver kiválasztása sem egyszerű feladat, hiszen ahány szoftver (és verzió), annyi különböző képesség és jellemző.

A gyakorlat során tehát az LPG nevezetű szoftvert fogjuk használni, mint olyat, amely a leginkább igazodik a laborgyakorlat célkitűzéseihez. Természetesen sok egyéb tervkészítő alkalmazás is rendelkezésre áll, számunkra azonban bőven elegendő a PDDL 2.2 kompatibilitás. Ezt az LPG-TD 1.0 biztosítja.

A PDDL nyelv verziószáma tehát igen fontos tényező, mivel a PDDL is folyamatosan fejlődik, módosul, bővül (a tervkészítő közösség igényeinek megfelelően). Hat főbb verziója ismeretes: 1.0, 1.7, 2.1, 2.2, 3.0, és 3.1. Mindegyik verziónak megvan a maga története, de főként arról ismertek, hogy rendre a 2 évente megrendezésre kerülő ***Nemzetközi Tervkészítési Verseny (International Planning Competition, IPC)*** hivatalos nyelvét képezik. Magyarán, a versenyen a nemzetközi tervkészítő közösség mindig az éppen aktuális verziószámú PDDL-ben leírt problémákon mérheti össze az erejét.

1998-ban az 1.0-ás (McDermott és társai, 1998), 2000-ben az 1.7-es (McDermott, 2000), 2002-ben a 2.1-es (Fox és Long, 2003), 2004-ben a 2.2-es (Edelkamp és Hoffmann, 2003), 2006-ban a 3.0-ás (Gerevini és Long, 2005), 2008-ban pedig a 3.1-es PDDL (Helmert, 2008) volt a verseny hivatalos nyelve. A 3.0-ás és PDDL temérdek új elemet tartalmaz elődeihez képest (aminek csak kis kiegészítése a 3.1-es). Ezek bemutatása azonban sajnos messze túlmutat a laborgyakorlat, s így e segédlet keretein. A 2.2-es változat viszont még épp elég aktuális és áttekinthető ahhoz, hogy a lényegét be tudjuk mutatni. A PDDL leírásnak (a 2.1-es változat óta) három szintje ismert:

I. **STRIPS-ES**
II. **NUMERIKUS**
III. **TEMPORÁLIS**

Az ***I. szint*** a problémák elsőrendű logikai leírását biztosítja (STRIPS-hez hasonló módon); a ***II. szint*** már nem csak logikai, hanem numerikus változókat és kényszereket is megenged, továbbá jósági kritériumok megfogalmazására is lehetőséget ad; a ***III. szint*** pedig már időzítéseket is tartalmaz. A 2.2-es PDDL-ben ezen felül lehetőség van még úgynevezett ***származtatott predikátumok (derived predicates)***, és ***időzített kezdeti literálok (timed initial literals)*** megadására is. Ezekre a későbbiekben még részletesen kitérünk (lásd. II.3.5, és II.3.6). Előtte azonban térjünk vissza a Sussman anomália megoldásához.

II.2.3. Sussman anomália egyfajta megoldása

Most, miután a II.2.2-es szakaszban foglaltaknak megfelelően kiválasztottuk, hogy a PDDL melyik verzióját, illetve ahhoz mérten mely tervkészítő eszközt fogjuk használni, rátérhetünk a II.2.1-es szakaszban informálisan vázolt problémára: a Sussman anomáliára.

Ahhoz, hogy a problémával bármit is kezdeni lehessen, először is le kell „fordítani" a tervkészítő eszköz, nevezetesen az LPG számára. Magyarán meg kell adnunk a probléma informális leírásának formális reprezentációját PDDL-ben. Ehhez először is azt kell tudnunk, hogy a PDDL nem egy domain-specifikus leírónyelv. Ez azt jelenti, hogy különböző ágens-környezetek/tárgyterületek különböző problémáinak leírására alkalmas – általános.

Ebből kifolyólag különválik benne a tervkészítési környezet (az ágenst is beleértve), illetve a konkrét megoldandó probléma leírása. Ennek a modularitásnak köszönhetően adott tervkészítési környezethez különálló módon konstruálhatunk különböző problémákat anélkül, hogy a környezet leírásán bármit is változtatni kellene.

Első lépésben tehát dekomponálnunk kell a II.2.1-es szakaszban vázolt problémát egy általános, probléma-független részre, és egy speciális, éppen aktuális esetet leíró részre. Az előbbit ***domain leírásnak***, míg az utóbbit ***probléma leírásnak*** nevezik.

Domain-leírás elkészítése

Mi a Sussman anomália probléma-független része? – merül fel a kérdés. A válasz nem egyértelmű – értelmezés kérdése. A probléma formalizálása nem egy mechanikus lépés. Nem adható rá általános algoritmus (egyelőre), csak néhány irányelv, melyek segítségével szisztematikusan, intuíciónkra hagyatkozva közelíthetünk a megoldáshoz.

Először is célszerű kiemelnünk a főbb fogalmakat (többnyire főneveket, igéket) az informális leírásból[3]. Miről is van szó? Adottak kockák, és adott egy asztal. A kockák lehetnek egymáson, és lehetnek az asztalon is, de valamelyiken biztosan. A kockákat – adott feltételek mellett – mozgatni lehet. Egyelőre ennyi.

Se a kockáknak, se az asztalnak nincs különösebb ismérve. Tehát, mint objektumoknak, nincsenek saját, különálló, egyedi jellemzőik, attribútumaik. Viszont adott közöttük egy reláció: a „*rajta*". Ez is tekinthető tulajdonságnak. Ezt kell tehát első nekifutásra megragadnunk.

Hívjuk segítségül az elsőrendű logikát, avagy más néven predikatív kalkulust! Vezessünk be egy alkalmas predikátumot arra, hogy reprezentálni tudjuk a „rajta" tulajdonságot/relációt! Legyen ez mondjuk az „`on/2`", ahol a 2-es szám a predikátum aritására, avagy argumentumainak számára vonatkozik (hiszen 2 objektum kapcsolatát kell leírnunk általa).

[3] Hasonlóan az UML (Unified Modelling Language) fél-formális módszertanához.

Tehát egyetlen pedikátumunk „`on(?A,?B)`” alakban adható meg általánosságban, ahol „`?A`” és „`?B`” logikai változókat jelöl, melyek az Univerzum (nyilván logikai értelemben) tetszőleges atomjával, avagy objektumával egyesíthetők. Egyelőre. A későbbiekben ugyanis kiköthetjük az említett változók osztályhoz tartozását (hogy csak az adott osztályból vehessék fel értéküket). Egyelőre azonban nincs szükségünk ilyesfajta megkötésekre, ilyesfajta **típusosság**ra.

Objektumainknak, változóinknak tehát egyelőre nincs típusa, mivel – szerencsére – nincs rá szükség ahhoz, hogy reprezentálni tudjuk a problémát. Minden objektum (az asztalt is beleértve) egyetlen osztályhoz tartozik, amit egyetlen tulajdonság jellemez csupán: az „`on/2`”. E tulajdonság adja meg tehát, hogy egy teszőleges objektum épp melyik másik objektumon van rajta.

Az áttekinthetőség érdekében törekednünk kell az egyszerűségre. Törekednünk kell arra, hogy a probléma minél egyszerűbb leírását tudjuk megadni (nem csak az „Ockham borotvája” elvből kifolyólag, hanem praktikus okokból is, például a későbbi hibajavítás, módosíthatóság, bővíthetőség, skálázhatóság megkönnyítése végett[4], stb).

Természetesen nem egyszerűsíthetjük a leírást minden határon túl, mivel előbb-utóbb információveszteséggel jár. Ilyen értelemben tehát célszerű kompromisszumra törekednünk a reprezentált információ mennyisége, pontosabban a leírás kifejezőereje és egyszerűsége közt.

Adott tehát egy predikátumunk, amivel a tudást megfelelő logikai tény-kijelentések formájában tudjuk reprezentálni. PDDL-ben ennek deklarációja a következőképp néz ki:

```
(define     (domain sussman1)
(:requirements :strips :equality)
(:constants       table)
(:predicates      (on ?x ?y)))
```

Tehát PDDL-ben a domain-leírás egy különálló szövegfájl adott szintaxissal (azaz nyelvtannal) és szemantikával (azaz annak megadásával, hogy melyik „mondat” mikor „igaz”). Jelen segédletben azonban ennek részleteire nem kívánunk kitérni, mivel egyrészt túlmutat a segédlet keretein, másrészt az irodalomban úgyis megtalálható.

Nézzük inkább a fenti PDDL domain-leírást! Ami elsőre szemet szúrhat, az a zárójelek tetemes száma. Ennek oka, hogy a PDDL nyelvet a LISP (LIS*t* P*rocessor*) mintájára dolgozták ki (McCarthy, 1960). Mivel a LISP egy funkcionális nyelv, ezért például a…

```
(+ a b)
```

…deklaráció jelentése „`a`” és „`b`” összege, ahol a „`+`” egy függvényt (funkciót, operátort) jelöl, aminek 2 bemenő paramétere/argumentuma van: „`a`” és „`b`”. Természetesen a PDDL nem funkcionális, hanem deklaratív, nincsenek benne klasszikus értelemben vett függvények,

[4] Ahogyan az áram (pl. a hallgatói áram) is mindig a gyengébb ellenállás irányába folyik… ☺

csak predikátumok. Viszont a LISP-hez hasonlóan ***prefix***, azaz mindenfajta predikátum a zárójelekkel körbezárt (és szóközökkel elválasztott) *listák* első eleme.

A fenti PDDL domain-leírás ilyen értelemben tehát egy lista, melynek első eleme egy „`define`" string. Ezt követi egy 2-elemű lista:

```
(domain sussman1)
```

Mivel domain-leírásról van szó, ezért ennek első eleme a „`domain`" string, második eleme viszont tetszőleges – a domain nevét jelöli. Esetünkben ez legyen „`sussman1`".

A következő elem újfent egy lista:

```
(:requirements :strips :equality)
```

A lista első eleme a „`:requirements`" string. Ez jelzi, hogy itt most a domain-leírás tervkészítőkkel szemben támasztott követelményeinek felsorolása következik. A lista többi eleme pedig az úgynevezett ***követelmény flag-ek*** halmaza.

Szerencsére esetünkben egyelőre elég a „`:strips`" és „`:equality`" követelmények megadása. Ezzel tehát azt deklaráljuk, hogy a domain összes problémája leírható egyszerű STRIPS-es formalizmus, és egyenlőségvizsgálat által. Tehát annak a tervkészítőnek, amely e domain problémáit kívánja megoldani, fel kell készülnie erre, azaz meg kell felelnie ezeknek a követelményeknek, tudnia kell olyan STRIPS-es problémákat megoldani, melyekben egyenlőségvizsgálat is előfordul.

A „`:requirements`" tehát igen fontos része a PDDL domain-leírásnak, hiszen jelzi a tervkészítők számára, hogy mire számítsanak. Amennyiben egy tervkészítő nincs felkészítve valamely követelményre, úgy hibát/figyelmeztetést dob, és legtöbbször nem ad megoldást a problémára. Szerencsére azonban esetünkben ez nemigen várható…

A domain-leírás következő eleme egy lista:

```
(:constants       table)
```

A konstansok deklarációja nem kötelező, csak akkor szükséges, ha létezik olyan objektum (ezt nevezik ***konstans***nak), amely a domain összes létező problémájában előfordul. Esetünkben van ilyen: az asztal. A domain-leírás következő sora:

```
(:predicates      (on ?x ?y))
```

A lista első eleme a „`:predicates`" string. Ez jelöli, hogy itt most a különböző predikátumok megadása következik. A lista többi eleme pedig egy-egy lista, ami rendre a különböző predikátumokat deklarálja. Esetünkben csak egy ilyen lista van, mivel csak egyetlen predikátumot kívánunk bevezetni:

```
(on ?x ?y)
```

Ez tehát az ominózus „`on`” predikátum PDDL-definíciója, ahol „`?x`” és „`?y`” a predikátum két argumentumát jelöli – két változót. A változók megnevezése tehát mindig „`?`”-jellel indul. A predikátum „olvasata” (angolul) a következő: *X on Y*. Magyarán azt jelöli/jelenti, hogy egy tetszőleges X egy tetszőleges Y-on van. A predikátumok definíciójában az argumentumok (amennyiben vannak) mindig és kötelezően változók, nem pedig konkrét objektumok.

Ezzel tehát eljutottunk a fenti domain-leírás végére, mindazonáltal még nem vagyunk kész. Valami hiányzik – valami fontos! Az informális leírásban ugyanis szerepel még a következő rész is (lásd. II.2.1): „…*pusztán csak a kockák rakosgatásával*.”

Ezek szerint tehát van egy ***cselekvés*** is, amit egyelőre még nem formalizáltunk. A PDDL természetesen erre is lehetőséget ad, hiszen mit érne a tervkészítés cselekvések nélkül?… Mielőtt azonban nekifognánk a teljes PDDL domain-leírás elkészítésének, meg kell tudnunk fogalmazni, hogy esetünkben, a Sussman anomália kapcsán, milyen cselekvések jöhetnek szóba, és mi jellemzi őket. Sőt, abban is meg kell állapodnunk, hogy általában mi jellemez egy-egy ilyen cselekvést!

A leírás szerint esetünkben csupán egyetlen cselekvésről beszélhetünk: „*áthelyezés*”. A következetesség jegyében jelöljük ezt az angol „`move`” szóval. A „`move`”-nak három ***paramétere*** van: MIT, HONNAN, HOVA. Azaz cselekvésünk olyan, mint valamiféle függvény – paraméterezett. Ahhoz, hogy végre lehessen hajtani, meg kell adnunk, hogy konkrétan MIT kívánunk áthelyezni, HONNAN, és végül HOVA.[5]

Ezen felül azt is definiálnunk kell, hogy mik a cselekvés ***logikai elő- és utó-feltételei***. Ésszerűen végiggondolva az előfeltételek (amik tehát a cselekvés végrehajtása előtt teljesülnek[6], amiknek igaznak kell lennie ahhoz, hogy a cselekvés „működjön”, azaz végrehajtásra kerüljön, és ne maradjon hatástalan) a következők:

- A kocka, amit át akarunk helyezni, ott van, ahonnan elvesszük.
- Nincs rajta más kocka.
- A kockát nem ugyanoda kívánjuk átrakni, mint ahonnan elvesszük.
- A kockát vagy az asztalra tesszük, vagy olyan kockára, amin nincs más kocka.

Tehát, amennyiben az előfeltételek teljesülnek, a cselekvés végrehajtható. Végrehajtását követően a következő állítások lesznek igazak:

- A kocka ott van, ahová raktuk.
- A kocka nincs ott, ahonnan elvettük.

[5] Természetesen elképzelhetők bemenő-paraméter nélküli cselekvések is.

[6] …itt a lényeg, hogy próbáljunk meg „deklaratívan gondolkozni”, ahogyan Szeredi Péter Tanár úr is sugallta „Deklaratív Programozás” c. tárgya során. Ne függvényekben gondolkozzunk, amiknek be- és kimenete van, hanem kijelentésekben, tényekben, igazságokban (amiknek nem része a következtetés)! Tehát például egy cselekvés előfeltételeinek meghatározásakor egyszerűen csak jelentsük ki, hogy mi igaz olyankor, és kész.

Körvonalaztuk tehát a „move” cselekvést. Megadtuk paramétereit, elő- és utófeltételeit. Bővítsük tehát ennek megfelelően eddigi domain-leírásunkat – reprezentáljuk a cselekvés(séma)t[7] PDDL nyelven:

```
(define     (domain sussman1)
(:requirements :strips :equality)
(:constants       table)
(:predicates      (on ?x ?y))

(:action move
:parameters       (?cube ?from ?to)
:precondition     (and  (on ?cube ?from)
                        (not  (exists (?x) (on ?x ?cube)))
                        (not  (= ?cube table))
                        (not  (= ?cube ?from))
                        (not  (= ?cube ?to))
                        (not  (= ?from ?to))
                        (or   (= ?to table)
                              (not  (exists (?y) (on ?y ?to)))))

:effect           (and  (on ?cube ?to)
                        (not (on ?cube ?from)))))
```

Látható tehát, hogy a „define”-nal kezdődő lista egy újabb elemmel bővült, ami lényegében egy újabb, viszonylag összetett lista. Ez szolgál a cselekvés(séma) leírására:

```
(:action move
:parameters       (?cube ?from ?to)
:precondition     (and  (on ?cube ?from)
                        (not  (exists (?x) (on ?x ?cube)))
                        (not  (= ?cube table))
                        (not  (= ?cube ?from))
                        (not  (= ?cube ?to))
                        (not  (= ?from ?to))
                        (or   (= ?to table)
                              (not  (exists (?y) (on ?y ?to)))))

:effect           (and  (on ?cube ?to)
                        (not  (on ?cube ?from))))
```

Tetszőleges számú ilyen lista-elem adható a domain-leíráshoz attól függően, hogy hány cselekvést kívánunk bevezetni. Esetünkben csupán csak egyetlen cselekvés jön szóba, ezért egyetlen ilyen lista-elem lesz.

A cselekvés leírás tehát egy lista, aminek első eleme az „:action” string. Második eleme a cselekvés megnevezése – esetünkben „move”. Harmadik eleme a „:parameters” string, amit egy – akár üres – lista követ, amely felsorolja a nevezett cselekvés bemenő paramétereit változók formájában. Esetünkben ezek a következők:

[7] Valójában csak ***cselekvés-sémáról*** van szó, hiszen a végrehajtás során, egy adott terv részeként, a változók adott behelyettesítése mellett beszélhetünk csak konkrét cselekvésről.

```
(?cube ?from ?to)
```

A „?cube” bemenő paraméter azonosítja a mozgatni kívánt kockát, a „?from” és a „?to” pedig rendre a kiindulási, és célpozíciót. A cselekvés-leírás következő eleme a „:precondition” string, amit egy lista követ, amely a cselekvés előfeltételének megfelelő (prefix jelölésű) elsőrendű logikai állítás. Lássuk ezt részletesen:

```
(and  (on ?cube ?from)
      (not  (exists (?x) (on ?x ?cube)))
      (not  (= ?cube table))
      (not  (= ?cube ?from))
      (not  (= ?cube ?to))
      (not  (= ?from ?to))
      (or   (= ?to table)
            (not  (exists (?y) (on ?y ?to)))))
```

Ez tehát egy elsőrendű logikai állítás – éppen 7 állítás ÉS-kapcsolata. Ennek első tagja:

```
(on ?cube ?from)
```

Mivel a cselekvés végrehajtásakor a bemenő paraméterek mindenképp teljesen behelyettesítve érkeznek, ezért a „?cube” és a „?from” változók be lesznek helyettesítve – konkrét objektumok lesznek a terv végrehajtása során. Épp azok, amik a bemeneten érkeztek. Ügyeljünk tehát erre a „láncolásra”!! Az ÉS-kapcsolat következő tagja:

```
(not (exists (?x) (on ?x ?cube)))
```

Ez tehát egy *egzisztenciális*an kvantifikált logikai állítás negáltja, amely kimondja, hogy nem létezik olyan „?x”, hogy (on ?x ?cube) teljesüljön. Ugyanez *univerzális* kvantorral a következő lenne:

```
(forall (?x) (not (on ?x ?cube)))
```

Ami újdonságot jelenthet itt, az a ***kvantifikált változók*** használata. Ez ugyanis az egyetlen módja annak, hogy a bemenő változókón túl, általában az aktuális világállapotra hivatkozzunk. Itt tehát nem olyan egyszerű a helyzet, mint mondjuk Prolog-ban (Colmerauer, 1975), ahol mintaillesztés útján dől el, hogy mi a változók értéke. **PDDL-ben** vagy – a cselekvések paraméter-részében lévő – **bemenő változókat,** vagy **konkrét objektumokat,** vagy **kvantifikált logikai változókat használunk arra, hogy** a cselekvések elő- és utófeltételeiben **az aktuális világ-állapotra hivatkozzunk.**

Az ÉS-kapcsolat következő 4 tagja:

```
(not  (= ?cube table))
(not  (= ?cube ?from))
(not  (= ?cube ?to))
(not  (= ?from ?to))
```

Ezekben már jól láthatóan egyenlőség-vizsgálat is szerepel[8]. A 4 feltétel valamivel többet mond, mint amit eredetileg specifikáltunk (hogy a kiindulási, és a célhely nem lehet azonos) [9]. Az előbbi 4 feltételből ugyanis az első 3 (amit ezidáig halványabban jelöltünk) csupán csak a robusztusságot szolgálja. Nincs semmi más funkciójuk. Csak azért van rájuk szükség, hogy – típusok hiányában – még véletlenül (pl. helytelenül megadott kiindulási tudásbázis esetén) se fordulhasson elő a „`move`" cselekvés értelmetlen bemenő-paraméterekkel történő meghívása/végrehajtása.

Itt jegyzem meg, hogy a jelenlegi LPG implementáció kényes az egyenlőség-vizsgálat argumentumainak sorrendjére. Példának okáért adott fentebb a következő:

```
(not   (= ?cube table))
```

Habár az egyenlőség-reláció elvben szimmetrikus, mégis amennyiben konstans objektumot (`table`) szerepltetnék a 0. helyen, és változót (`?cube`) az 1-diken, úgy az LPG hibát jelezne. Erre legyünk tehát tekintettel a gyakorlat során…

Egyébként az egyenlőség-vizsgálat használatára hívja fel a figyelmet a domain-leírás követelmény részében az „`:equality`" flag. Igazság szerint ez a feltétel elhagyható volna, ahogyan a többi követelményt sem feltétlen szükséges megadni. Alkalmazás-függő, hogy melyik-mennyire veszi komolyan. Az LPG példának okáért a követelményektől kvázi függetlenül parse-olja fel a domain- és probléma-leírást, és ennek alapján dönti el, hogy valójában milyen követelményeknek kell megfelelnie a problémamegoldás során. Mindazonáltal a `:requirements` rész – redundáns jellege ellenére – irányadó lehet mind a tervkészítő számára, mind a forrást böngésző Olvasó számára, így helyes használatát az elkövetkezőkben (is) megköveteljük.

Az egyenlőség-vizsgálat (`=`), ÉS-kapcsolat (`and`), és negálás (`not`) mellett természetesen még más egyéb logikai operátorok is rendelkezésünkre állnak ahhoz, hogy az előbbieknél összetettebb logikai állításokat is meg tudjunk fogalmazni. Ilyen például a VAGY-kapcsolat (`or`), és az implikáció (`imply`). Amíg az előbbi legalább 2-argumentumú, addig az utóbbi már csakis csak 2-argumentumú logikai művelet.

Az ÉS-kapcsolat utolsó, 7-dik eleme a következő:

```
(or    (= ?to table)
       (not   (exists (?y) (on ?y ?to))))
```

[8] Látható, hogy ez nem numerikus egyenlőséget vizsgál, hanem logikai értelemben vett *egyesíthetőséget*.

[9] Ennek oka, hogy a specifikáció nem veszi figyelembe az implementáció sajátosságait. A gyakorlatban ez igen gyakori, és legtöbbször elkerülhetetlen. Oka, hogy a *specifikáció* per definitio magasabb absztrakciós szintű, általánosabb, mint megvalósítása, az *implementáció*. Viszont a specifikációnak ennek ellenére törekednie kell a konzisztenciára, a teljességre, és arra, hogy a benne foglalt elvek változatlanul, esetleg némi szükséges kiegészítéssel kerülhessenek megvalósításra. Ennek ellenére legtöbbször igen nehéz, sőt reménytelen verifikálni a kettő közötti megfelelést, garantálni a minőséget, stb. Főleg szabványosan! Például a Windows-os LPG egy CygWin nevezetű Linux-emulációs környezet alatt fut (lásd. http://cygwin.com/), és amennyiben a CygWin nem kezeli le valamely LPG által dobott kivételt, úgy az LPG, pontosabban a CygWin elszáll (pl. egy STATUS_ACCESS_VIOLATION hibaüzenet kíséretében).

Ez egy újabb lista – két állítás VAGY-kapcsolata. Ezek szerint VAGY az igaz, hogy a célpozíció az asztal, VAGY pedig nincs olyan „`?y`”, amely a célpozíción rajta van. Ezt akár így is írhatnánk:

```
(imply      (not (= ?to table))
            (not  (exists (?y) (on ?y ?to))))
```

Implikációként felírva valamelyest „olvashatóbb”. Ezek szerint tehát, ha nem az asztalra kívánjuk rakni a kockát, akkor azon a valamin, ahová rakjuk (ami már biztosan kocka), nem lehet semmi. Az asztalon pedig nyilván több kocka is lehet egyszerre.

Visszatérve a ***bemenő paraméterek***re: úgy érdemes elképzelni őket, mint amik mindig teljesen behelyettesítve érkeznek. Az előfeltételeket úgy, mint amik illeszkednek a világ aktuális állapotára. Az utófeltételekre a következők vonatkoznak:

Az utófeltételeket egy „`:effect`” string előzi meg a cselekvés leírásában. Ezt követi a cselekvés következményét leíró elsőrendű logikai állítás. Esetünkben, a „`move`” cselekvés kapcsán ez a következő:

```
(and (on ?cube ?to)
     (not (on ?cube ?from)))
```

FIGYELEM: a cselekvések következmény-részében kizárólag konjunkció foglalhat helyet, a diszjunkciónak nincs értelme. Ennek megfelelően tehát a „`move`” cselekvés következmény-része két állítás konjunkciója. Ebből az első:

```
(on ?cube ?to)
```

A cselekvés végrehajtása után (kizárólag akkor, ha az előfeltételek teljesültek) igaz lesz tehát, hogy az átmozgatni kívánt kocka a célhelyen van. Másrészt igaz lesz az is, hogy:

```
(not (on ?cube ?from))
```

Azaz nem lesz igaz, hogy az átmozgatni kívánt kocka a kiindulási pozícióban van. Ezekre az utófeltételekre tehát úgy kell tekintenünk, mint amik módosítják a világ állapotát. Alapértelmezésben ugyanis a ***zárt-világ feltételezés***sel élünk, amely kimondja, hogy amiről nem tudjuk, hogy igaz-e, azt hamisnak tekintjük. Ezek szerint a negált következmények eltávolítják a tudásbázisból a tényt (retract), míg a ponált következmények hozzáadják (assert).

Probléma-leírás elkészítése

Az előbbiekben, a Sussman anomália kapcsán áttekintettük a PDDL probléma-független, ún. domain-leírási mechanizmusának alapjait. Megadtuk a Sussman anomália probléma-független, PDDL leírását. A megfelelő szövegfájlt, amely ezt a leírást tartalmazza, nevezzük a következőképp: `feladat1a_domain1.pddl`

A következőkben a Sussman anomália probléma-függő részét, az úgynevezett probléma-leírást fogjuk elkészíteni. Ez a leírás tehát az előbbi domain-leíráshoz fog igazodni, és a következő fájlnév alatt mentjük el[10]: `feladat1a_domain1_problem1.pddl`

A probléma-leírás tehát, mint már említettük, igazodik a kapcsolódó domain-leíráshoz. Legegyszerűbb esetben 3 fő részből tevődik össze (a fejlécen kívül):

1. Világban létező véges sok objektum felsorolása
2. Világ kiinduló-állapotában igaz tények felsorolása
3. Célállapot meghatározása

Mielőtt azonban rátérünk a PDDL-szintaxis részleteire, gondoljuk végig, hogy az előbbi, probléma-független leírásnak megfelelően mik lesznek esetünkben a probléma-függő elemek! Tegyünk – egyelőre informálisan – eleget az előbbi három kitételnek!

A II-2. ábra szerint négy objektumunk van: 3 kocka, és 1 asztal. Nevezzük ezeket rendre „a”-nak, „b”-nek, „c”-nek, és „`table`”-nek. Ebből a „`table`” konstans – mindig, minden probléma esetén szerepel, ezért már definiáltuk a domain-leírásban. A probléma-leírásban felesleges újra szerepltetni. Lássuk tehát, hogy mi igaz a kiinduló-állapotban!

A II-2. ábra szerint „c” az „a”-n van, „a” viszont az asztalon, azaz a „`table`”-en. Emellett még az is igaz, hogy „b” a „`table`”-en van. Összességében tehát három tény lesz igaz a Sussman anomália kiinduló-állapotában:

```
(on c a)
(on a table)
(on b table)
```

FIGYELEM: a STRIPS kapcsán bevezetett zárt világ feltételezéssel élve csupán csak az igaz tények felsorolása szükséges – ami nincs megadva, azt hamisnak tekintjük. Az LPG ennek megfelelően hibát jelez, ha negált tényállítást talál a kezdetben igaz tények közt.

Végül hasonlóképp határozzuk meg azt, hogy minek kell teljesülnie a célállapotban. Itt nyilván már nem csak tények konjunkciója szerepelhet, hanem tetszőleges elsőrendű (akár kvantifikált[11]) logikai állítás. Esetünkben ez viszonylag egyszerű: „a” van „b”-n, „b” van „c”-n, „c” pedig az asztalon, azaz a „`table`”-en. Ez a cél. Ennek kell teljesülnie végül. Ez PDDL-ben a következő állításnak felel meg:

```
(and  (on a b)
      (on b c)
      (on c table))
```

[10] Érdemes valamiféle elnevezési konvenciót/szisztémát követnünk ahhoz, hogy ne kuszálódjanak össze a különböző PDDL leírások.

[11] Azaz olyat, amelyben egzisztenciális és/vagy univerzális kvantor szerepel. Mindazonáltal az LPG jelen verziója sajnos nem támogatja ezt egynél több kvantifikálható objektum esetén, így használata felesleges.

Összefoglalva, a Sussman anomália PDDL-alapú probléma-leírása a következő:

```
(define (problem sussman1_1)
(:domain sussman1)
(:objects a b c)
(:init      (on c a)
            (on a table)
            (on b table))
(:goal      (and  (on a b)
                  (on b c)
                  (on c table))))
```

A PDDL-alapú probléma-leírás tehát szintaxisában hasonló a domain-leíráshoz. Egyetlen egy LISP-es listából áll, aminek első eleme egy „`define`” string. Második eleme egy 2-elemű lista, aminek első eleme a „`problem`” string, második eleme pedig a probléma tetszőleges megnevezése. Esetünkben ez „`sussman1_1`”. Ezt követi a domain-hivatkozás:

```
(:domain sussman1)
```

Vegyük észre, hogy itt nem fájl-szinten, hanem azonosító-szinten történik a hivatkozás (hiszen a domain-t még fentebb „`sussman1`”-nek neveztünk el). Ezután következik a probléma-specifikus objektumok felsorolása:

```
(:objects a b c)
```

Itt nyilván azért nem szerepel a „`table`”, mivel már a domain-leírásban definiáltuk, mint olyan konstans objektumot, amely minden Sussman domain-hez tartozó probléma esetén megjelenik. Az objektumok megadását a kezdetben igaz – teljesen behelyettesített (!!!) – tények felsorolása követi:

```
(:init      (on c a)
            (on a table)
            (on b table))
```

Végül pedig a célállapot meghatározása:

```
(:goal      (and  (on a b)
                  (on b c)
                  (on c table)))
```

A célállapotot egyébként nyilván nem kell teljes egészében specifikálni. Lehet ugyanis több cél(világ)állapot, több olyan tény-elrendezés, amelyben teljesül a megoldástól elvárt feltétel. Elég tehát egy olyan logikai állítás megadása, amely csak és kizárólag a célállapotokban teljesül. Ezzel – félig-meddig implicite – a kívánatos világ-állapotok (ún. szituációk) halmazát definiáljuk.

A probléma-leírás végére értünk, következhet a probléma megoldása!

Probléma megoldása

Az előbbiekben megkonstruáltuk a Sussman anomália PDDL nyelvű domain- és probléma-leírását. Vizsgáljuk meg, miképpen futtathatjuk ezen választott tervkészítő alkalmazásunkat, az LPG-t![12]

Az LPG futtatására több mód is van, mi azonban ezek közül egyelőre csak azt az esetet vizsgáljuk, amikor Windows Command Prompt, azaz parancssor alól történik a futtatás. Először is lépjünk be az LPG könyvtárába (pl. `c:\lpg`). Bizonyosodjunk meg arról, hogy az előbb létrehozott `feladat1a_domain1.pddl`, és `feladat1a_domain1_problem1.pddl` fájlok rendelkezésre állnak (pl. magában a `c:\lpg` könyvtárban). Az LPG futtatásához ekkor a következőt írjuk be:

```
lpg-td-1.0 -o feladat1a_domain1.pddl -f feladat1a_domain1_problem1.pddl -quality
```

Amennyiben mindent helyesen csináltunk, úgy a futás eredményeképpen a megoldás (különböző egyéb információkkal egyetemben) kiíródik mind a képernyőre, mind egy külön fájlba. A kigenerált fájl neve: `plan_feladat1a_domain1_problem1.pddl_1.SOL`

Látható, hogy a kigenerált fájl nevét az LPG automatikusan határozza meg a probléma-leírást tartalmazó PDDL-fájl neve alapján. Vegyük szemügyre a fentebb szereplő parancssori utasítást, amivel az LPG-t futásra bírtuk.

Jól láthatóan 3 opciót adtunk meg:

1. A „`-o`” arra való, hogy megadjuk a domain-leírást tartalmazó PDDL-fájl elérését (az úgynevezett operátor-fájlt, amely tehát az operátorokat, pontosabban operátor-sémákat tartalmazza).
2. A második opció a „`-f`”. Ez arra való, hogy megadjuk a probléma-leírást tartalmazó PDDL-fájl elérését (az úgynevezett tényeket (facts) tartalmazó fájlt).
3. Végül a harmadik opció a „`-quality`”. Ennek az opciónak nincs paramétere – arra való, hogy az LPG modalitását szabályozza.

Az LPG jelen verziójának kétféle modalitása van: „`-quality`”, és „`-speed`”. Az előbbi hatására jobb minőségű, rövidebb tervek adódnak valamivel hosszabb idő alatt, míg az utóbbi esetben a futási idő, nem pedig a terv minősége az elsődleges szempont.[13]

A domain-leírás, probléma-leírás, és modalitás megadása szükséges ahhoz, hogy a LPG fusson. A modalitás helyett esetleg még a „`-n`” opcióval az is megadható, hogy a problémának legfeljebb hány megoldására vagyunk kíváncsiak.

Térjünk most azonban még egy kicsit vissza az előbbi futás eredményéhez. A képernyőre nagyjából a következő kerül:

[12] A laborgyakorlat hardware-software hátteréhez mérten egyelőre csak a Windows-os esetet vizsgáljuk.

[13] Érdemes megfigyelni, hogy amíg „`-quality`” opcióval többszörös meghívásra is általában ugyanazt a tervet adja a tervkészítő, addig „`-speed`” opcióval meghívva hívásonként más-más lehet az eredmény.

```
Parsing domain file:  domain 'SUSSMAN1' defined ... done.
Parsing problem file:  problem 'SUSSMAN1_1' defined ... done.



Modality: Quality Planner

Number of actions             :      64
Number of conditional actions :       0
Number of facts               :      32


Analyzing Planning Problem:
      Temporal Planning Problem: NO
      Numeric Planning Problem: NO
      Problem with Timed Initial Litearals: NO
      Problem with Derived Predicates: NO

Evaluation function weights:
     Action duration 0.00; Action cost 1.00

Computing mutex... done

Preprocessing total time: 0.06 seconds

Searching ('.' = every 50 search steps):
 Restart.
.. search limit exceeded. Restart.
.. search limit exceeded. Restart.
..

Plan computed:
   Time: (ACTION) [action Duration; action Cost]
 0.0003: (MOVE C A TABLE) [D:1.0000; C:1.0000]
 1.0005: (MOVE B TABLE C) [D:1.0000; C:1.0000]
 2.0008: (MOVE A TABLE B) [D:1.0000; C:1.0000]

Solution found:
Total time:      0.78
Search time:     0.03
Actions:         3
Execution cost:  3.00
Duration:        3.000
Plan quality:    3.000
     Plan file:       plan_feladat1a_domain1_problem1.pddl.SOL



Do not use option -quality for better solutions.
```

Itt először is az látszik, ahogyan az LPG beolvassa (be-parse-olja) a domain- és probléma-leírást tartalmazó fájlokat. Ezt követi a modalitás jelzése. Jelen esetben ugyebár „`-quality`” opcióval hívtuk meg a tervkészítőt, így a modalitás „`Quality Planner`”. A következő három sor a cselekvések, és a tények számát adja meg.

Elsőre meglepő lehet, hogy miért éppen 64 cselekvést említ az LPG, mikor mi csak egyetlen egyet adtunk meg. Ennek oka egyszerű: nem egy cselekvést, hanem egy *cselekvés-sémát* adtunk meg. A cselekvés-séma 3 bemenő paramétert tüntet fel. A problém-leírás szerint pedig összesen 4 különböző objektum van a világban. Ezek szerint összesen 4^3=64 különböző bemenettel hívható meg a „`move`” cselekvés.

A következő sor azt jelzi, hogy egyetlen feltételes cselekvésünk (`conditional action`) sincs. Feltételes cselekvések azok, melyek következmény-része feltételes (lásd. IV.1.1: FRRT). A PDDL már a kezdetek óta alkalmas ilyesfajta cselekvések leírására, mi azonban a kapcsolódó laborgyakorlat alkalmával nem foglalkozunk velük, mivel nem egyeztethető össze azzal a célkitűzésünkkel, amely szerint kizárólag csak klasszikus tervkészítéssel foglalkozunk.[14]

A következő sor szerint 32 tény van. Ez nyilván nem a kiindulási tudásbázisban (vagy munkamemóriában) szereplő tények száma, ami – mint fentebb látszik – pusztán csak 3. Nem, ez itt az összes elképzelhető tény számát adja meg, amit úgy kapunk, hogy vesszük egyetlen 2-argumentumú predikátumunkat, és minden lehetséges módon behelyettesítjük: így adódik 4^2=16 darab különböző lehetséges ponált tény[15], amelynek elvben (pl. a zárt-világ feltételezés elhagyásával) még másik 16 negált párja lehet. Így összesen 16+16=32 különböző lehetséges tényről beszélhetünk.

A következő pár sor a probléma analízisét mutatja. Mivel esetünkben (egyelőre) semmi különöset sem alkalmaztunk a domain és/vagy a probléma leírásában, ezért minden szempontnál a „`NO`” szerepel.

Az LPG a keresés során használ egy úgynevezett kiértékelő függvényt (evaluation function), amely súlyozza a cselekvéseket végrehajtási idejük, és áruk alapján. Jelen esetben – alapértelmezésben – ezek a súlyozási faktorok 0 és 1 értékűek.

A következő sor szerint sikeresen kiszámításra kerültek az úgynevezett „***mutex***”-ek. A mutex-szócska az angol „**mut**ually **ex**clusive” szavak összetételéből adódik, és olyan cselekvésekre vonatkozik, melyek kölcsönösen kizárják egymást (bővebben lásd. IV.1).

A mutex-ek alatt az előfeldolgozás ideje, a tervkészítés/keresés folyamata, és maga a megoldás, azon belül is a terv lépései láthatók. Minden lépés mellett ott szerepel, hogy melyik időpillanatban indul meg végrehajtása, mennyi ideig tart, és mennyibe kerül. Alapértelmezésben (a nem temporális, STRIPS-es tervkészítésnél) minden időtartam és ár egységnyi.[16]

[14] Továbbá sajnos az LPG jelen verziója sem támogatja…

[15] Amiknek jó része nyilván abszurdum, a tervkészítő azonban „buta” módon ezt mégsem veszi észre. Hasonlóan, a cselekvések esetében is látható volt, hogy nem történt szűrés arra vonatkozólag, hogy mely paraméter-kombinációkkal nem lesz sohasem (értsd. semmilyen világállapot mellett sem) végrehajtható a cselekvés. Mi azonban tudjuk, hogy például az összes olyan bemeneti paraméter-kombináció elhagyható volna, amelyben valamely bemeneti elem legalább kétszer szerepel. Mindennek a „least commitment”, avagy a ***legkisebb megkötés elve*** az oka. Ez az elv allegorikusan itatja át a részben rendezett tervkészítést.

[16] Úgy látszik az LPG ehhez az időtartamhoz a biztonság kedvéért még hozzáad 0.2-0.3 microsec.-ot.

Végül a tervkészítés ideje, és a megoldás(ok) paraméterei látható(k). Jelen esetben egy 3 lépés hosszú tervet sikerült találni, amely 3 időegység alatt kivitelezhető, illetve költsége és minősége egyaránt 3.[17]

A futás során kigenerált fájl (esetünkben: `plan_feladat1a_domain1_problem1.pddl_1.SOL`) is lényegében ugyanezeket az információkat tartalmazza, csak valamivel felületesebb formában. A lényeg maga a megoldás. Nézzük most ezt:

```
0:   (MOVE C A TABLE) [1]
1:   (MOVE B TABLE C) [1]
2:   (MOVE A TABLE B) [1]
```

Ezek szerint a 0. időpillanatban a C kockát az A-ról az ASZTAL-ra kezdjük el áthelyezni, majd közvetlen ez után, az 1. időpillanatban a B kockát az ASZTAL-ról a C-re tesszük, és végül, a 2. időpillanatban az A kockát az ASZTAL-ról a B kockára rakjuk át. Ezzel tehát megoldottuk a Sussman anomáliát.

Nyilván a Sussman anomália esetében „ránézésre” is látszik a megoldás. Látszik tehát, hogy az LPG által visszaadott terv optimális: nincs ennél gyorsabb, hatékonyabb megoldás. Viszont ez az egyszerűség ne tévesszen meg minket – „gazdag egyszerűség”-ről van szó ugyanis! Gondoljunk csak bele: 3 kocka esetén még nyilván könnyen átlátható és megoldható a probléma. De *mi van akkor, ha több kocka van? Ha 5, 7, 9, vagy esetleg 11 kocka van az asztalon?* (lásd. IV.3).

Szerencsére a problémák skálázása (mint már említettük) nem jár a domain-leírás megváltoztatásával. A leírás ugyanolyan egyszerű marad, mint volt. Egyedül a probléma-leírás gazdagodik újabb objektumokkal, tényekkel, és esetleg célra vonatkozó feltétellel. Ha valami okból mindez mégis komplikáltnak tűnne, ne rettenjünk vissza! Csak az első néhány alkalommal van ez így. Valójában nagyon egyszerű, intuitív, és kézenfekvő. Gyakorlatlanságunk folytán elveszhetünk az apró részletekben, amiket később már rutinszerűen elsajátítunk. „Szoknia kell a szemnek”…

Minden esetre remélhető, hogy a fentiek elolvasása után nem maradt bennünk számottevő kérdés a Sussman anomália PDDL-ben történő reprezentációjával, és annak LPG általi megoldásával kapcsolatban. A következőkben ugyanis – egy komolyabb, „életszagú” példa kapcsán – szükségünk lesz mindarra, amit az imént tanultunk.

Megjegyzés: a szemfüles Olvasó esetleg észrevehette, hogy a fentebb bemutatott domain leírást egy újabb predikátum bevezetésével bizonyos fokig le lehet „egyszerűsíteni”. Egy újabb (`free/1`) predikátum bevezetésével – amit csak a mozgatható kockákra mondunk ki – a domain leírás cselekvés-sémáiból kiiktathatók az egzisztenciális kvantorok, ami amellett, hogy „egyszerűsíti” a leírást, jelentősen gyorsítja a probléma megoldását. Viszont ennek az „egyszerűsítésnek” ára van: 1 cselekvés-séma helyett immár 2 cselekvés-séma szükséges, továbbá a probléma-leírásban is több kezdeti tény szerepel (lásd. IV. 4).[18]

[17] A megoldás minősége és költsége az LPG sajátja – keresési heurisztikáihoz szükséges faktorok.
[18] …ez is csak azt bizonyítja, hogy még ilyen egyszerű problémáknak is több ekvivalens reprezentációja lehet.

II.3. Tervkészítési mintapélda

Az előbbiekben még csak sejtettük a mai determinisztikus tervkészítő rendszerek képességeit, most azonban egy komolyabb, életszerűbb példán is bemutatjuk őket. Ennek megfelelően az előző (lásd. II.2.3) szakaszra alapozva kidolgozásra kerül egy összetett mintapélda, amelynek során megismerkedhetünk a típusos, numerikus, és temporális PDDL leírással, illetve az így leírt problémák megoldásával.

A (fiktív) történet a következő: megbízást kapunk a NASA-tól, hogy adjunk egy általános, lehetőleg optimális megoldást a Cape Canaveral-ban található Műhold Vezérlő Központ (SCC – Satellite Control Center) számára az alábbi ütemezési probléma kapcsán:

Műholdak egy csoportjának adott megfigyelési műveleteket kell végezni. Minden megfigyelési művelet kapcsán adott, hogy mely csillagászati célpontról milyen típusú felvételt kell készíteni (pl. termográfia, spektrográfia). A műholdak különféle mérőműszereket szállítanak. Más-más műszerek más-más mérési üzemmódokat támogatnak, azaz más-más típusú felvétel készítésére alkalmasak. A műholdak által végezhető műveletek:

Fordulás (`turn-to`):	*A műhold egy adott célpont felé fordul.*
Bekapcsolás (`switch-on`):	*Adott műszer bekapcsolása a műholdon.*
Kikapcsolás (`switch-off`):	*Adott műszer kikapcsolása a műholdon.*
Kalibrálás (`calibrate`):	*Bekapcsolt műszer kalibrálása, ami csak akkor hajtható végre, ha a műhold épp egy alkalmas kalibrációs célpont felé néz.*
Fényképezés (`take-image`):	*A műhold valamely műszere adott üzemmódban felvételt készít arról a célpontról, amire a műhold éppen néz. Nyilván csak akkor hajtható végre, ha a műszer be van kapcsolva, és előzőleg kalibrálva volt.*

Nyilván több megközelítése/megoldása is volna a fentebb informálisan vázolt problémának. Esetünkben azonban nem kérdés – tervkészítési eszközökkel oldjuk meg.

II.3.1. Egyszerű STRIPS-es leírás

Az előbbi informális leírás PDDL reprezentációjának előállítása lényegében a II.2.3 szakaszban leírtak mintájára történik. Ezért most – a redundancia elkerülése érdekében – nem térünk ki az előzőleg ismertetett részletekre, hanem inkább az új, kihívást jelentő szempontokra próbáljuk helyezni a hangsúlyt.

Tehát adott a fenti probléma-leírás. Próbáljuk meg – eddigi ismereteink alapján – megadni ennek PDDL leírását! Először is a probléma-független rész kiragadásával célszerű kezdenünk, majd azon belül megfogalmaznunk a konkrét problémát. A fenti leírásból azonban egyelőre még nem tűnik ki, hogy mi lesz a konkrét probléma. Nincsenek benne

konkrétumok. Ennek oka egyszerű: a NASA arra kért minket, hogy a fenti leírásnak megfelelő tetszőleges problémára általában tudjunk megoldást adni.

Domain-leírás

Tehát egyelőre próbáljuk meg megfogalmazni a domain-leírást. Vezessük be a predikátumokat, amik az objektumok jellemzőit/tulajdonságait, illetve a köztük lehetséges kapcsolatokat (relációkat) írják le.

Legyen először is egy `(on_board ?i ?s)` predikátumunk, amely akkor igaz, ha az „`?i`” műszer – példány (!!) – az „`?s`” műholdon van. Legyen egy `(supports ?i ?m)` predikátumunk, amely akkor igaz, ha az „`?i`” műszer támogatja az „`?m`” üzemmódot. Legyen egy `(pointing ?s ?d)` predikátumunk, amely akkor igaz, ha az „`?s`” műhold a „`?d`” célpont felé néz. A valósághűség kedvéért[19] legyen még egy `(power_avail ?s)` predikátumunk is, amely akkor igaz, ha az „`?s`” műholdon van feles energia-ellátás. Legyen továbbá egy `(power_on ?i)` predikátumunk, amely akkor igaz, ha az „`?i`” műszer be van kapcsolva. Hasonlóan legyen egy `(calibrated ?i)` predikátumunk is, amely akkor igaz, ha az „`?i`” műszer be van kalibrálva. Ehhez szükséges még egy `(calibration_target ?i ?d)` predikátum is, amely akkor igaz, ha az „`?i`” műszer kalibrálásához a „`?d`” célpont felé kell néznie a műholdnak. Végül legyen egy `(have_image ?d ?m)` predikátumunk is, amely akkor igaz, ha a „`?d`” célpontról valamely műhold már készített „`?m`” üzemmódban képet. Összefoglalva:

```
(:predicates
         (on_board ?i ?s)
         (supports ?i ?m)
         (pointing ?s ?d)
         (power_avail ?s)
         (power_on ?i)
         (calibrated ?i)
         (have_image ?d ?m)
         (calibration_target ?i ?d))
```

Miután sikerrel megragadtuk a probléma kapcsán szóban forgó predikátumokat, kíséreljük meg megadni a lehetséges cselekvések sémáit! Kezdjük máris a „`turn-to`” cselekvéssel: ennek 3 paramétere lesz: (1) a műhold(objektum/példány), amely a mozgást végzi, (2) a célpont ahová fordul, illetve (3) ahonnan elfordul.

```
:parameters       (?s ?d_new ?d_prev)
```

Ezek tehát azok az információk, amik szükségesek ahhoz, hogy a műveletet végre lehessen hajtani.
Ezek után térjünk rá a „`turn-to`” cselekvés előfeltételeinek, és következményeinek vizsgálatára: előfeltétel, hogy a műhold a cselekvés megkezdésekor a régi célpont felé nézzen, és az új célpont ne egyezzen meg a régivel.

[19] …mondjuk a NASA-val folytatott előzetes egyeztetések (követelmény-specifikáció) nyomán…

```
:precondition      (and  (pointing ?s ?d_prev)
                         (not (= ?d_new ?d_prev)))
```

A cselekvés hatására a műhold az új célpont felé néz, nem pedig a régi felé:

```
:effect            (and  (pointing ?s ?d_new)
                         (not (pointing ?s ?d_prev)))
```

Összefoglalva, a „`turn-to`” cselekvés(séma) PDDL leírása a következő:

```
(:action turn_to
   :parameters     (?s ?d_new ?d_prev)
   :precondition   (and  (pointing ?s ?d_prev)
                         (not (= ?d_new ?d_prev)))
   :effect         (and  (pointing ?s ?d_new)
                         (not (pointing ?s ?d_prev)))
)
```

Haladjunk tehát sorra végig a cselekvéseken, és rendre adjuk meg PDDL-leírásuk! Végeredményben mondjuk a következő domain-leírást kapjuk:

```
(define (domain satellite)
 (:requirements :strips :equality)
 (:predicates (on_board ?i ?s)
              (supports ?i ?m)
              (pointing ?s ?d)
              (power_avail ?s)
              (power_on ?i)
              (calibrated ?i)
              (have_image ?d ?m)
              (calibration_target ?i ?d))

 (:action turn_to
   :parameters       (?s ?d_new ?d_prev)
   :precondition     (and  (pointing ?s ?d_prev)
                           (not (= ?d_new ?d_prev)))
   :effect           (and  (pointing ?s ?d_new)
                           (not (pointing ?s ?d_prev)))
 )

  (:action switch_on
   :parameters       (?i ?s)
   :precondition     (and  (on_board ?i ?s)
                           (power_avail ?s))
   :effect           (and  (power_on ?i)
                           (not (calibrated ?i))
                           (not (power_avail ?s)))
  )

  (:action switch_off
   :parameters       (?i ?s)
   :precondition     (and  (on_board ?i ?s)
                           (power_on ?i))
   :effect           (and  (not (power_on ?i))
                           (power_avail ?s))
  )

  (:action calibrate
```

```
    :parameters       (?s ?i ?d)
    :precondition     (and   (on_board ?i ?s)
                      (calibration_target ?i ?d)
                             (pointing ?s ?d)
                             (power_on ?i))
    :effect                  (calibrated ?i)
   )

   (:action take_image
    :parameters       (?s ?d ?i ?m)
    :precondition     (and   (calibrated ?i)
                             (on_board ?i ?s)
                             (supports ?i ?m)
                             (power_on ?i)
                             (pointing ?s ?d))
    :effect                  (have_image ?d ?m)
   )
)
```

A kapott domain-leírásban tehát semmi új sincs ahhoz képest, mint amit a II.2.3-as szakaszban láttunk. Viszont itt jegyezném meg, hogy amennyiben kommentezni szeretnénk a forráskódot (domain- vagy probléma-leírást), úgy a sor elejére „;” pontosvesszőt tegyünk.

Probléma-leírás

Az előbbiekben elkészítettük a Műhold-ütemezési probléma STRIPS-es PDDL domain-leírását. Furcsa módon azonban a probléma informális leírásában nem találtunk olyan konkrétumot, amely definiálná, hogy mi a konkrét megoldandó probléma. Ennek okát fentebb már megindokoltuk. Ahhoz tehát, hogy tesztelni tudjuk a fenti domain-leírást, magunknak kell valamiféle konkrét problémát kigondolni, és PDDL-ben reprezentálni. Legyen ez a következő:

Legyen adott 1 műhold:	`satellite0`
Legyen adott 1 műszer:	`instrument0`
Legyen adott 3-féle üzemmód:	`image1, spectrograph2, thermograph0`
Legyen adott 7 csillagászati célpont:	`Star0, GroundStation1, GroundStation2, Phenomenon3, Phenomenon4, Star5, Phenomenon6`

A PDDL probléma-leírásban tehát az objektumok felsorolásánál a következő szerepel:

```
(:objects
      satellite0
      instrument0
      image1
      spectrograph2
      thermograph0
      Star0
      GroundStation1
      GroundStation2
      Phenomenon3
      Phenomenon4
      Star5
      Phenomenon6)
```

Ezek után a domain-leírásban szereplő predikátumok, és az imént bevezetett objektumok segítségével adjuk meg a kiindulási világállapotot (azaz a kezdetben igaz tények halmazát):

- Legyen igaz, hogy „`instrument0`” támogatja a „`thermograph0`” üzemmódot.
- Legyen igaz, hogy „`instrument0`” bekalibrálásához a műholdnak a „`GroundStation2`” célpont felé kell néznie.
- Legyen igaz, hogy „`instrument0`” a „`satellite0`” műhold fedélzetén van.
- Legyen igaz, hogy „`satellite0`” műhold számára biztosított az energia-ellátás.
- Legyen igaz, hogy „`satellite0`” a „`Phenomenon6`” célpont felé néz.

Ennek megfelelően a PDDL-es probléma-leírásban a következő rész fog szerepelni:

```
(:init
      (supports instrument0 thermograph0)
      (calibration_target instrument0 GroundStation2)
      (on_board instrument0 satellite0)
      (power_avail satellite0)
      (pointing satellite0 Phenomenon6))
```

Miután a probléma-leírás kapcsán megadtuk a lehetséges objektum(példány)okat, illetve a kiindulási világ-állapotot, nyilván specifikálnunk kellene a célállapotot is. A célállapotban legyen igaz, hogy…

- …a „`Phenomenon4`” célpontról készült felvétel „`thermograph0`” üzemmódban, és
- a „`Star5`” célpontról készült felvétel „`thermograph0`” üzemmódban, és
- a „`Phenomenon6`” célpontról is készült felvétel „`thermograph0`” üzemmódban.

Ennek megfelelően a PDDL-es probléma-leírásban a következő rész szerepel:

```
(:goal (and (have_image Phenomenon4 thermograph0)
            (have_image Star5 thermograph0)
            (have_image Phenomenon6 thermograph0)))
```

Ezzel tehát teljes egészében definiáltunk egy konkrét problémát az előzőleg definiált domain-en belül. Lássuk tehát e probléma teljes PDDL leírását:

```
(define (problem satellite1)
```

```
        (:domain satellite)
        (:objects
              satellite0
              instrument0
              image1
              spectrograph2
              thermograph0
              Star0
              GroundStation1
              GroundStation2
              Phenomenon3
              Phenomenon4
              Star5
              Phenomenon6)

        (:init
              (supports instrument0 thermograph0)
              (calibration_target instrument0 GroundStation2)
              (on_board instrument0 satellite0)
              (power_avail satellite0)
              (pointing satellite0 Phenomenon6))

        (:goal (and (have_image Phenomenon4 thermograph0)
                    (have_image Star5 thermograph0)
                    (have_image Phenomenon6 thermograph0)))
)
```

Probléma-megoldás

Az előbbiekben definiált problémát a hozzá tartozó domain-leírással együtt immár az LPG bemenetére adhatjuk, hogy kiderüljön van-e egyáltalán megoldása, és ha igen, akkor vajon micsoda. Az LPG a következő megoldást adja az előbbi problémára:

```
0:   (SWITCH_ON INSTRUMENT0 SATELLITE0) [1]
0:   (TURN_TO SATELLITE0 GROUNDSTATION2 PHENOMENON6) [1]
1:   (CALIBRATE SATELLITE0 INSTRUMENT0 GROUNDSTATION2) [1]
2:   (TURN_TO SATELLITE0 PHENOMENON4 GROUNDSTATION2) [1]
3:   (TAKE_IMAGE SATELLITE0 PHENOMENON4 INSTRUMENT0 THERMOGRAPH0) [1]
4:   (TURN_TO SATELLITE0 STAR5 PHENOMENON4) [1]
5:   (TAKE_IMAGE SATELLITE0 STAR5 INSTRUMENT0 THERMOGRAPH0) [1]
6:   (TURN_TO SATELLITE0 PHENOMENON6 STAR5) [1]
7:   (TAKE_IMAGE SATELLITE0 PHENOMENON6 INSTRUMENT0 THERMOGRAPH0) [1]
```

Értelmezzük a kapott tervet: a 0. időpillanatban két művelet párhuzamos végrehajtása történik. Egyrészt bekapcsolásra kerül az „`instrument0`” műszer a „`satellite0`” műhold fedélzetén, másrészt eközben a műhold elfordul „`Phenomenon6`” felől „`GroundStation2`” felé. Az előbbi lépés oka nyilván az, hogy amíg a műszer nincs bekapcsolva, addig nem lehet vele felvételt készíteni, nekünk márpedig az a célunk…; az utóbbi lépés oka pedig nyilván az, hogy felvételt csak kalibrált műszerrel lehet készíteni, amihez be kell kalibrálni, ami – a műszertől függően – csak adott irányban lehetséges.

Miután végbement a bekapcsolás és elfordulás, az 1. időpillanatban elkezdődik a műszer kalibrálása. Ez a 2. időpillanatban fejeződik be, mikor is a műhold „`Phenomenon4`” célpont

felé kezd fordulni. Ennek oka, hogy – a célállapot specifikációja szerint – erről a célpontról kell, hogy legyen egy „`thermograph0`” üzemmódban készült képünk. Szerencsére az „`instrument0`” műszer támogatja ezt az üzemmódot, így – más műszer nem lévén – nem kizárt a felvétel elkészítése.

A 3. időpillanatban már a „`Phenomenon4`” célpont felé néz egyetlen műholdunk. Ekkor tehát el is készítjük a megfelelő felvételt, majd – a felvétel elkészítését követő – 4. időpillanatban a „`Star5`” célpont felé fordulunk. Az 5. időpillanatban erről is készítünk egy megfelelő felvételt, majd a 6. időpillanatban a „`Phenomenon6`” célpont felé fordulunk, és a 7. időpillanatban erről is elkészítjük a megfelelő felvételt. Ezzel tehát a 8. időpillanatra 9 lépésben elkészültünk. Ráadásul ennél hatékonyabb megoldás nincs is a jelen problémára, így tehát kijelenthetjük, hogy a kapott megoldás optimális.

II.3.2. Típusok bevezetése

Az előbbiekben esetleg zavaró lehetett, hogy „egy lapon” említettünk minden objektumot: műszert, műholdat, üzemmódot, stb. Valóban, számunkra ez még csak zavaró, de a tervkészítő számára még akár megtévesztő is lehet, ugyanis ilyenkor temérdek olyan ténnyel és cselekvéssel kell számolnia, ami esetleg értelmetlen, sohasem fog előfordulni. Például, mivel a `supports/2` predikátumnál nincs deklarálva, hogy első argumentuma valamiféle műszer, második argumentuma pedig valamiféle üzemmód, ezért a tervkészítő a tervkészítés során `(supports thermograph0 instrument0)` jellegű tényekre is fel lehet készülve, sőt, általánosságban fel is készül.

Namármost nyilvánvaló, hogy ilyen biztosan nem fog előfordulni – abszurdum. Hasonlóan a tervkészítő számításba veheti a `turn_to(Phenomenon4, thermograph0, instrument0)` konkrét cselekvést, ami nyilván hasonlóképp abszurd, mint az előző példa. A tervkészítő mégis számol(hat) vele, hiszen az objektumok között semmi különbséget nem tud tenni.

Az ilyen, és ehhez hasonló esetek számának csökkentése, illetve egyéb problémák elkerülése érdekében szükséges lehet specifikusabbá tenni a PDDL leírást. Ezt eddig (pl. a Sussman anomália kapcsán) vagy előfeltételekbe iktatott explicit egyenlőség-vizsgálatokkal (lásd. II.2.3), vagy – mint az előző szakaszban (lásd. II.3.1) – a kezdeti világ-állapot megfelelő megadásával oldottuk meg. A két módszer kiegészítette egymást, ám még így együtt sem voltak igazán megfelelők.

Egyrészt a cselekvések előfeltételeinek bővítése *átláthatatlanná teheti* a cselekvések leírását, hiszen a bemenő paraméterek számával négyzetesen arányos számú egyenlőség-vizsgálatra lehet szükség (nem is beszélve a különböző konstansokkal történő esetleges összehasonlításokról).

Másrészt a kiindulási tudásbázis helyes megadása *túl implicit* megkötés. Esetünkben, a műholdas példánál, szerencsére viszonylag kézenfekvő volt, hogy minek kell az adott probléma kapcsán kezdetben igaznak lennie, és a tervkészítő szerencsére nem is ment tévútra. Viszont egy összetettebb probléma esetén már ilyesmi is előfordulhat!

A megoldást, mint ahogyan a szakasz címe is utal rá, a típusok bevezetése jelenti. Gyorsabb, egyszerűbb, és megbízhatóbb. Mostantól tehát minden objektumnak és változónak lesz típusa. A típusra gondolhatunk úgy, mint valamiféle tulajdonságra, ami minden objektumhoz rendel egy típus-azonosítót, de úgy is, mint valamiféle kategóriára, osztályra, amelyhez az objektum tartozik. Személy szerint én inkább ezt a szemléletet tartom célravezetőbbnek. Ekkor ugyanis úgy tekintünk a (logikai értelemben vett) univerzum atomjaira, a lehetséges objektumokra, mint egy halmaz elemeire. A halmaznak pedig vannak részhalmazai: ezek a „leszármazottak” – az egyes „osztályok”.

Az osztályokat, mint kategóriákat, nyilván úgy határozzuk meg, hogy olyan elemeket tartalmazzon, amelyek azonos tulajdonságokkal rendelkeznek (még ha értékük el is tér egymástól). Ezeket az elemeket tehát képzeletünkben „körbekarikázzuk”, és osztálynak nevezzük. Az említett elemek, az objektumok ekkor az *osztály példányai*. Az osztályra pedig úgy is tekinthetünk, mint ***típus***ra.

A típusok, ahogyan az osztályok is, hierarchiát alkotnak. Adottak szülő-típusok, és gyermek-típusok. Az utóbbiak öröklik az előbbiek tulajdonságait, jellemzőit. Pontosabban (mivel PDDL-ben nem tulajdonságok által implicite, hanem explicite adjuk meg a típusokat) a gyermek-típusú objektumok szülő-típusúak is egyben.

Az ős-típus PDDL-ben az „`object`”, ami nem csak fizikai objektumokra vonatkozhat, hanem – az O*ntology* W*eb* L*anguage*-ben (OWL, lásd. McGuinness és van Hermelen, 2004) ismeretes „Thing” ős-osztályhoz hasonlóan – bármire.[20]

Domain-leírás

Egészítsük tehát ki az előbbi domain- és probléma-leírást típusokkal! Kezdjük a domain-leírás kiegészítésével:

[20] Mindazonáltal szerencsésebbnek tartom az OWL-es „Thing” megnevezést, mivel intuitívebb. Gondoljuk csak meg, mi van akkor, ha PDDL-ben reprezentálni szeretnénk pl. az iskolai végzettséget. Legyen mondjuk „általános”, „középszintű”, „felsőfokú”, és „doktori”. Ezeket – akár konstans – objektumokként kell deklarálnunk. Típusuk az „iskolai végzettség”. Viszont ez a típus alapértelmezésben leszármazottja az „`object`”-nek, s így az előbb felsorolt végzettségek is rendre mind-mind ilyen típusúak lesznek: objektumok. Mindazonáltal zavaró lehet, mikor pl. a felsőfokú végzettségre, mint fogalomra, objektumként tekintünk, nemde? – Természetesen nem fizikai, hanem *logikai objektum*ot kell értenünk alatta…

```
(define (domain satellite)
       (:requirements :strips :equality :typing)
       (:types satellite direction instrument mode)
       (:predicates
              (on_board ?i - instrument ?s - satellite)
              (supports ?i - instrument ?m - mode)
              (pointing ?s - satellite ?d - direction)
              (power_avail ?s - satellite)
              (power_on ?i - instrument)
              (calibrated ?i - instrument)
              (have_image ?d - direction ?m - mode)
              (calibration_target ?i - instrument ?d - direction))

  (:action turn_to
   :parameters         (?s - satellite ?d_new ?d_prev - direction)
   :precondition       (and    (pointing ?s ?d_prev)
                               (not (= ?d_new ?d_prev)))
   :effect                     (and  (pointing ?s ?d_new)
                               (not (pointing ?s ?d_prev)))
  )

  (:action switch_on
   :parameters         (?i - instrument ?s - satellite)
   :precondition       (and    (on_board ?i ?s)
                               (power_avail ?s))
   :effect             (and    (power_on ?i)
                               (not (calibrated ?i))
                               (not (power_avail ?s)))
  )

  (:action switch_off
   :parameters         (?i - instrument ?s - satellite)
   :precondition       (and    (on_board ?i ?s)
                               (power_on ?i) )
   :effect             (and    (not (power_on ?i))
                               (power_avail ?s))
  )

  (:action calibrate
   :parameters         (?s - satellite ?i - instrument ?d - direction)
   :precondition       (and    (on_board ?i ?s)
                               (calibration_target ?i ?d)
                               (pointing ?s ?d)
                               (power_on ?i))
   :effect             (calibrated ?i)
  )

  (:action take_image
   :parameters         (?s - satellite ?d - direction ?i - instrument ?m - mode)
   :precondition       (and    (calibrated ?i)
                               (on_board ?i ?s)
                               (supports ?i ?m)
                               (power_on ?i)
                               (pointing ?s ?d)
                               (power_on ?i))
   :effect             (have_image ?d ?m)
  )
)
```

Az első különbség, ami szembetűnik, hogy a `:requirements` közt immár a `:typing` flag is szerepel, jelezve a tervkészítő számára, hogy ebben a domain-ben az objektumok domain-specifikus típussal is rendelkezhetnek. Eddig mindvégig, kimondva-kimondatlanul az „`object`” őstípust tekintettük az objektumok típusának.

A másik különbség, ami az előző, II.3.1-es szakaszban bemutatott domain-leíráshoz képest feltűnik, hogy közvetlenül a követelmények deklarációja után következik a lehetséges típusok felsorolása. Ezek az „`object`”-en felül definiált, egyedi típusaink:

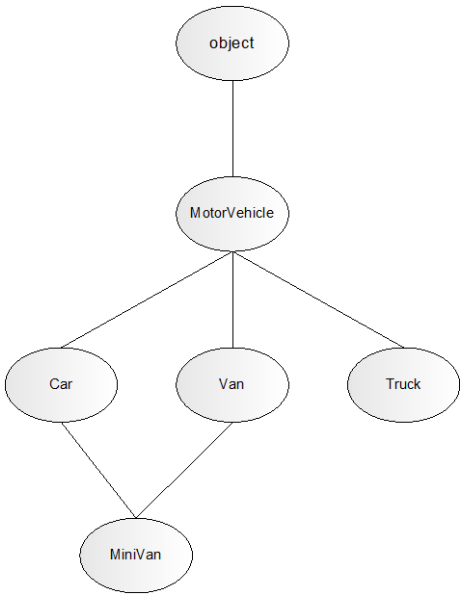
```
(:types satellite direction instrument mode)
```

Az eddigiekhez mérten nyilván nem meglepő, hogy éppen erről a 4 típusról van szó. Jelentésük nyilvánvaló. Mindnyájan az „`object`” típus (és önmaguk[21]) leszármazottai. A továbbiakban különbséget csak a konstansok, a predikátumok, a cselekvések bemenő-paramétereinek, és esetleg a kvantifikált változók megadásában látunk. Itt ugyanis, ahol eddig pusztán csak egyetlen tényező (objektum, vagy változó) szerepelt, immár a típus megadása is jelen lesz.

Amennyiben egymás után több azonos típusú tényezőt is fel szeretnénk sorolni (pl. egy cselekvés-séma bemenő paramétereinél több azonos típusú változót), úgy a PDDL kényelmi szolgáltatásként lehetővé teszi, hogy ne kelljen mindannyiszor kiírnunk a típust, elég a következő formában megadnunk:

```
(... ?x1 ?x2 ... ?xN - typex ?y1 ?y2 ... ?yM - typey ...)
```

Jelenlegi domain-leírásunkban nincs típus-hierarchia, mivel eddig még nem volt rá szükség. Típusaink alapértelmezésben mind az „`object`” őstípus leszármazottai – azonos szinten helyezkednek el az egyszintes típus-hierarchiában, és mind diszjunktak.[22] Mindazonáltal egy röpke pillanatra, a példa kedvéért alakítsunk ki egy ennél összetettebb típus-hierarchiát, amin bemutathatjuk az alapelveket és szintaxist.



II-3. ábra: objektum-típusok egy hierarchiája

[21] …mivel a típus-leszármazás reláció *reflexív*.
[22] PDDL-ben egy típus különböző altípusai alapértelmezésben diszjunktak, metszetük üres.

A II-3. ábrán jól látható a példa gyanánt felhozott típus-hierarchia. Az „`object`” őstípus mellett 5 egyénileg definiált típust vezetünk be: a „`MotorVehicle`” típus az „`object`” őstípus leszármazottja, és motoros gépjárműveket takar. A motoros gépjárműveket a példa kedvéért 3 különálló részre osztjuk: „`Car`”, amely a személygépkocsi kategóriájú járműveket takarja; „`Van`”, amely furgonokat takar; és „`Truck`”, amely a teherautókat takarja. Végül adott a „`MiniVan`” típus, amely a személyszállításra alkalmas, kisméretű furgonokat takarja, s így mind a „`Car`”, mind a „`Van`” típusnak leszármazottja.

Az imént bemutatott típus-hierarchia PDDL-ben a következő:

```
(:types Car Van Truck - MotorVehicle MiniVan - (either Car Van))
```

Látható tehát, hogy ahogyan eddig a változók és objektumok típusát, úgy adjuk most meg a típusok szülő-típusát is. Látható továbbá, hogy például a „`MotorVehicle`” típusnak nincs megadva a szűlő-típusa, ami ezért alapértelmezésben „`object`”.

Az említett típusok (`object`, `MotorVehicle`, `Car`, `Van`, `Truck`, `MiniVan`) mind úgynevezett ***elemi típusok***. Viszont a típus-deklarációban valami más is szerepel:

```
(either Car Van)
```

Ez egy úgynevezett összetett típus. Az ***összetett típusok*** elemi típusok úniójaként adódnak, és nincs saját egyedi azonosítójuk. Egy lista jelöli őket, aminek első eleme az „`either`” string, többi – legalább egy – eleme pedig rendre az unióban szereplő elemi típusok azonosítója.

Egy összetett típust akkor tekintünk egy másik – akár elemi, akár összetett – típus *leszármazott*jának, ha a benne szereplő összes elemi típus leszármazottja a másik típusnak. Egy összetett típust akkor tekintünk egy másik – akár elemi, akár összetett – típus *felmenő*jének, ha a másik típus az összetett típusban szereplő valamely elemi típus(ok) leszármazottja.

FIGYELEM: ha a domain-leírásban egy konstansnak, egy bemenő paraméter-változónak, vagy egy kvantifikált változónak nincs külön feltüntetve a típusa, úgy alapértelmezésben „`object`”.

Ezzel tehát megadtuk a típusok szintaktikáját és szemantikáját, viszont nem indokoltuk meg a hierarchia szükségességét. *Mi értelme van a típus-hierarchia bevezetésének, ha így megbonyolítja a szintaxist?* – merülhet fel a kérdés.

A válasz elég összetett, ezért egyelőre legyen számunkra elég annyi, hogy típus-hierarchia nélkül bizonyos esetekben kénytelenek lennénk a predikátumok, illetve cselekvések számát jelentősen megnövelni. Tegyük fel ugyanis, hogy adott valahány típusunk, amik nincsenek hierarchikus viszonyban egymással. Minden objektumnak van egy ilyen típusa. Ekkor ahhoz, hogy egy predikátum, vagy egy cselekvés több különböző típusú objektumra is „működjön”, kénytelenek lennénk típusok szerint duplikálni…

Probléma-leírás

Az előzőekben bemutattuk a típusok fogalmát, és azt, hogy miként vezethetjük be őket a domain-leírásba. A probléma-leírásban egyedül az objektumok[23], és a `:goal` részben szereplő, esetleges kvantifikált változók kapcsán szükséges bevezetni őket:

```
(define (problem satellite2)
      (:domain satellite)
      (:objects
            satellite0 - satellite
            instrument0 - instrument
            image1 spectrograph2 thermograph0 - mode
            Star0 GroundStation1 GroundStation2
            Phenomenon3 Phenomenon4 Star5
            Phenomenon6 - direction)
      (:init
            (supports instrument0 thermograph0)
            (calibration_target instrument0 GroundStation2)
            (on_board instrument0 satellite0)
            (power_avail satellite0)
            (pointing satellite0 Phenomenon6))
      (:goal (and (have_image Phenomenon4 thermograph0)
                  (have_image Star5 thermograph0)
                  (have_image Phenomenon6 thermograph0)))
)
```

Ezek után a tervkészítő a tervkészítés során már nem fog olyan objektumokkal kísérletezni pl. a cselekvések bemenő paramétereiben, melyeknek nem megfelelő a típusa. Tehát végső soron a típusok bevezetése egyfajta „gyorsító hatású” kényszerűség.

Probléma-megoldás

A típusok bevezetése önmagában azonban nem változtat a probléma megoldásán:[24]

```
0:   (SWITCH_ON INSTRUMENT0 SATELLITE0) [1]
0:   (TURN_TO SATELLITE0 GROUNDSTATION2 PHENOMENON6) [1]
1:   (CALIBRATE SATELLITE0 INSTRUMENT0 GROUNDSTATION2) [1]
2:   (TURN_TO SATELLITE0 PHENOMENON4 GROUNDSTATION2) [1]
3:   (TAKE_IMAGE SATELLITE0 PHENOMENON4 INSTRUMENT0 THERMOGRAPH0) [1]
4:   (TURN_TO SATELLITE0 STAR5 PHENOMENON4) [1]
5:   (TAKE_IMAGE SATELLITE0 STAR5 INSTRUMENT0 THERMOGRAPH0) [1]
6:   (TURN_TO SATELLITE0 PHENOMENON6 STAR5) [1]
7:   (TAKE_IMAGE SATELLITE0 PHENOMENON6 INSTRUMENT0 THERMOGRAPH0) [1]
```

[23] Sajnos az LPG jelen verziója nem támogatja az objektumok összetett típusának megadását. Ez azonban nem nagy akadály, hiszen az összetett típus megfelelő altípusának bevezetésével mindez kiváltható.

[24] Ha mindent alaposan végiggondoltunk, akkor ennek elvben így kell lennie. Ha a típusok bevezetése mégis szűkíti a megoldások körét, úgy vagy a típusok bevezetése helytelen, vagy a predikátumok, cselekvések értelmezése/szemantikája nem stimmel.

II.3.3. Numerikus változók bevezetése

A probléma eddigi PDDL reprezentációja színtisztán logikai leírásra szorítkozott. Mindazonáltal a valóságban a probléma ennél nyilvánvalóan bonyolultabb. A műhold fordulása időt vesz igénybe, és üzemanyag fogyasztással jár, továbbá a felvételek tárolása adatkapacitást igényel, stb. Ennek megfelelően egészítsük ki eddigi leírásunkat – tegyük inkább valósághűvé! A kiegészítés informálisan a következő:

„A műholdak üzemanyagkészlete és memóriája korlátos. A célpontok közötti fordulás a művelet idejével arányos mennyiségű üzemanyagot fogyaszt, az egyes felvételek eltárolásához pedig adott mennyiségű szabad memória szükséges.”

Ezek szerint nem csak új fogalmak bevezetésére lesz szükség (pl. üzemanyag, adatméret, fordulási idő), hanem numerikus korlátok (pl. csak olyan fordulás hajtható végre, ami nem igényel több üzemanyagot annál, mint ami rendelkezésre áll) betartása is. Ez utóbbiak nyilván a cselekvések előfeltételeiben fognak helyet foglalni.

Domain-leírás

Lássuk először is egy olyan PDDL domain-leírást, amely már a fentebb felsorolt szempontokat is magába foglalja:

```
(define (domain satellite)
        (:requirements :strips :typing :equality :fluents)
        (:types satellite direction instrument mode)
        (:predicates
                (on_board ?i - instrument ?s - satellite)
                (supports ?i - instrument ?m - mode)
                (pointing ?s - satellite ?d - direction)
                (power_avail ?s - satellite)
                (power_on ?i - instrument)
                (calibrated ?i - instrument)
                (have_image ?d - direction ?m - mode)
                (calibration_target ?i - instrument ?d - direction))
        (:functions
                (data_capacity ?s - satellite)
                (data ?d - direction ?m - mode)
                (slew_time ?a ?b - direction)
                (data-stored)
                (fuel ?s - satellite)
                (fuel-used))

        (:action turn_to
                :parameters     (?s - satellite ?d_new ?d_prev - direction)
                :precondition   (and    (pointing ?s ?d_prev)
                                        (not (= ?d_new ?d_prev))
                                        (>= (fuel ?s) (slew_time ?d_new ?d_prev)))
                :effect         (and    (pointing ?s ?d_new)
                                        (not (pointing ?s ?d_prev))
                                        (decrease (fuel ?s) (slew_time ?d_new ?d_prev))
                                        (increase (fuel-used) (slew_time ?d_new ?d_prev)))
        )
```

```
        (:action switch_on
                :parameters     (?i - instrument ?s - satellite)
                :precondition   (and    (on_board ?i ?s)
                                        (power_avail ?s))
                :effect         (and    (power_on ?i)
                                        (not (calibrated ?i))
                                        (not (power_avail ?s)))
        )

        (:action switch_off
                :parameters     (?i - instrument ?s - satellite)
                :precondition   (and    (on_board ?i ?s)
                                        (power_on ?i))
                :effect         (and    (not (power_on ?i))
                                        (power_avail ?s))
        )

        (:action calibrate
                :parameters     (?s - satellite ?i - instrument ?d - direction)
                :precondition   (and    (on_board ?i ?s)
                                        (calibration_target ?i ?d)
                                        (pointing ?s ?d)
                                        (power_on ?i))
                :effect         (calibrated ?i)
        )

        (:action take_image
                :parameters     (?s - satellite ?d - direction ?i - instrument ?m - mode)
                :precondition   (and    (calibrated ?i)
                                        (on_board ?i ?s)
                                        (supports ?i ?m)
                                        (power_on ?i)
                                        (pointing ?s ?d)
                                        (power_on ?i)
                                        (>= (data_capacity ?s) (data ?d ?m)))
                :effect         (and    (decrease (data_capacity ?s) (data ?d ?m))
                                        (have_image ?d ?m)
                                        (increase (data-stored) (data ?d ?m)))
        )
)
```

A bemutatott domain-leírásban pirossal külön kiemeltük, hogy az előző leíráshoz (lásd. II.3.2) képest mik a fő különbségek. Vegyük most ezeket sorra!

Először is látható, hogy a leírás elején, a követelményeket felsoroló `:requirements` részben immár egy újabb, `:fluents` flag-et is szerepeltetünk azért, hogy jelezzük a tervkészítő alkalmazások számára, hogy a probléma megoldásához numerikus változók kezelésére is szükség lehet.

A második, sokkal szembetűnőbb különbség a következő:

```
        (:functions
                (data_capacity ?s - satellite)
                (data ?d - direction ?m - mode)
                (slew_time ?a ?b - direction)
                (data-stored)
                (fuel ?s - satellite)
                (fuel-used))
```

Ez a rész a numerikus változók deklarációja. Azért szerepel az elején `:functions`, mert a PDDL-t specifikáló szakemberek döntése az volt, hogy numerikus értékű függvények formájában definiálják a numerikus változókat. Tehát minden egyes függvény egy-egy numerikus változónak felel meg. Pontosabban egy bizonyos „fajta” numerikus változónak.

Vegyük például a `(data_capacity ?s - satellite)` függvényt. Ez adott „`?s`” bemenetre valamilyen numerikus értéket ad eredményül. Ideális esetben minden „`satellite`” típusú objektum felett definiálva, azaz értelmezve van az értéke. Ebben az esetben tehát a `data_capacity` függvény a bemenetén kapott tetszőleges konkrét műhold-objektumhoz hozzárendeli annak adatkapacitását. ***Ezt*** a konkrét értéket tekintjük ***numerikus változó***nak, nem pedig magát a függvényt, vagy annak deklarációját!

Látható, hogy a definícióban nem csak 1-argumentumú, hanem több-argumentumú függvény (pl. `data, slew_time`) is szerepel. De vannak olyan függvények is (pl. `data-stored, fuel-used`), melyeknek egy argumentuma sincs. Ez utóbbi „függvények”, argumentum híján, végső soron már akár numerikus változóknak is tekinthetők.

Kicsit „fordított gondolkodásra” vall az egész: arról van szó ugyanis, hogy mi magunk, a tervkészítés során, a kezdeti állapot meghatározásakor adjuk meg, hogy adott bemenetre mi legyen a függvény értéke. Tehát a fenti függvényeknek nincs „képlete”, ami alapján a bementből adódik a kimenet. Nem, mi határozzuk meg a kimenetet. Mi adjuk meg, hogy egy-egy bemenethez egy-egy függvény milyen numerikus kimenő értéket társítson. Mintha csak numerikus változóknak adnánk értéket… – Erről van szó.

Így tehát pl. a `satellite0` objektum adatkapacitásának értékét tartalmazó numerikus változóra `(data_capacity satellite0)` formában hivatkozhatunk. Ez a jelölés talán még „beszédesebb” is, mint a szokványos.

A következő különbség, ami szembetűnik a domain-leírásban, a cselekvések elő- és utófeltételeiben látható. Haladjunk sorra: a „`turn-to`” cselekvés előfeltételei kiegészültek a következő feltétellel:

```
(>= (fuel ?s) (slew_time ?d_new ?d_prev)))
```

Ezek szerint ahhoz, hogy a fordulást véghez lehessen vinni, az „`?s`” műholdnak még legalább annyi üzemanyagra van szükséges, mint amennyi ideig a fordulás tart „`?d_prev`” célpont felől „`?d_new`” célpont felé. Ez a kényszer hozza tehát összefüggésbe az üzemanyag fogyasztást a fordulási idővel. Ezek szerint a két mennyiség egymással egyenesen arányos, ráadásul az egyszerűség kedvéért 1:1 arányban.

A „`turn-to`” cselekvésnek azonban nem csak az előfeltételei, hanem a következményei is kiegészültek:

```
(decrease (fuel ?s) (slew_time ?d_new ?d_prev))
(increase (fuel-used) (slew_time ?d_new ?d_prev)))
```

Ezek szerint tehát a „`turn-to`” cselekvés sikeres végrehajtása után az „`?s`” műhold üzemanyaga, azaz `(fuel ?s)` éppen `(slew_time ?d_new ?d_prev)` értékével csökken (ami az előbbi érvelés nyomán teljesen érthető). Ráadásul még a `fuel-used` függvény, pontosabban az összes műhold által felhasznált összes üzemanyag mennyiségét megadó numerikus változó értéke is épp ugyanennyivel nő.

Az „>=”, „=”, és „<=” beépített eljárások tehát függvények esetén nem érték-adásra, hanem érték-összehasonlításra szolgálnak. Érték-adásra/módosításra ekkor a „decrease”, „assign”, „increase”, „scale-up”, és „scale-down” beépített eljárások szolgálnak. Ekkor tehát meg kell adni, hogy minek mi legyen az értéke, vagy mi mennyivel nőjön/csökkenjen, szorzódjon/osztódjon. Mind az öt műveletnek 2-2 argumentuma van.

A következő, és egyben utolsó különbség (az előző domain-leíráshoz képest) a „take_image” cselekvés elő- és utófeltételeiben figyelhető meg. Az előfeltételek a következővel egészültek ki:

```
(>= (data_capacity ?s) (data ?d ?m)))
```

Ez a feltétel egyáltalán nem meglepő. Jelentése az, hogy az „?s” műhold még hátralévő adatkapacitása nagyobb kell, hogy legyen, mint azon adatmennyiség, amely a „?d” célpontról „?m” üzemmódban készült felvétel tárolásához szükséges.

A „take_image” cselekvés utófeltételei a következőkkel bővültek:

```
(decrease (data_capacity ?s) (data ?d ?m))
(increase (data-stored) (data ?d ?m)))
```

Ez a két – mellékhatással járó – következmény igen hasonló azokhoz, amiket előbb a „turn-to” cselekvés esetében figyelhettünk meg. Bár itt most nem üzemanyag-kapacitásról, hanem éppenséggel adat-kapacitásról van szó. Az első következmény tehát a műhold még meglévő üres tárkapacitását csökkenti annyival, amennyi helyet a „?d” célpontról „?m” üzemmódban készült felvétel tárolása igényel. A második következmény pedig éppen ennyivel növeli meg az összes műholdon letárolt összes eddig adat mennyiségét (data-stored).[25]

Probléma-leírás

Látható, hogy az előbbi domain-leírásnak megfelelő konkrét problémáknak várhatóan igen sok megoldása lehet. Ezért célszerű valamiféle irányadó ***jósági mérce*** bevezetése, amely leszűkíti a jó megoldások körét. Ezentúl tehát a megoldásnak már nem csak a numerikus korlátokat kell betartania, hanem lehetőség szerint még ezt a jósági mércét is célszerű minél inkább optimalizálnia (értsd. minimalizálnia, vagy maximalizálnia).

Az egyszerűség kedvéért tehát adjuk meg az előző, II.3.2-es szakaszban tárgyalt problémát kiegészítve a megfelelő függvények kezdeti értékével, és a jósági mércével:

```
(define (problem satellite3)
(:domain satellite)
(:objects
        satellite0 - satellite
        instrument0 - instrument
        image1 - mode
        spectrograph2 - mode
        thermograph0 - mode
```

[25] Gondoljunk esetleg arra, hogy a műholdak az elkészült felvételeket nem csak lokálisan tárolják, hanem leküldik az SCC-nek is.

```
        Star0 - direction
        GroundStation1 - direction
        GroundStation2 - direction
        Phenomenon3 - direction
        Phenomenon4 - direction
        Star5 - direction
        Phenomenon6 - direction
)
(:init
        (supports instrument0 thermograph0)
        (calibration_target instrument0 GroundStation2)
        (on_board instrument0 satellite0)
        (power_avail satellite0)
        (pointing satellite0 Phenomenon6)
        (= (data_capacity satellite0) 1000)
        (= (fuel satellite0) 112)
        (= (data Phenomenon3 image1) 22)
        (= (data Phenomenon4 image1) 120)
        (= (data Star5 image1) 203)
        (= (data Phenomenon6 image1) 144)
        (= (data Phenomenon3 spectrograph2) 125)
        (= (data Phenomenon4 spectrograph2) 196)
        (= (data Star5 spectrograph2) 68)
        (= (data Phenomenon6 spectrograph2) 174)
        (= (data Phenomenon3 thermograph0) 136)
        (= (data Phenomenon4 thermograph0) 134)
        (= (data Star5 thermograph0) 273)
        (= (data Phenomenon6 thermograph0) 219)
        (= (slew_time GroundStation1 Star0) 18.17)
        (= (slew_time Star0 GroundStation1) 18.17)
        (= (slew_time GroundStation2 Star0) 38.61)
        (= (slew_time Star0 GroundStation2) 38.61)
        (= (slew_time GroundStation2 GroundStation1) 68.04)
        (= (slew_time GroundStation1 GroundStation2) 68.04)
        (= (slew_time Phenomenon3 Star0) 14.29)
        (= (slew_time Star0 Phenomenon3) 14.29)
        (= (slew_time Phenomenon3 GroundStation1) 89.48)
        (= (slew_time GroundStation1 Phenomenon3) 89.48)
        (= (slew_time Phenomenon3 GroundStation2) 33.94)
        (= (slew_time GroundStation2 Phenomenon3) 33.94)
        (= (slew_time Phenomenon4 Star0) 35.01)
        (= (slew_time Star0 Phenomenon4) 35.01)
        (= (slew_time Phenomenon4 GroundStation1) 31.79)
        (= (slew_time GroundStation1 Phenomenon4) 31.79)
        (= (slew_time Phenomenon4 GroundStation2) 39.73)
        (= (slew_time GroundStation2 Phenomenon4) 39.73)
        (= (slew_time Phenomenon4 Phenomenon3) 25.72)
        (= (slew_time Phenomenon3 Phenomenon4) 25.72)
        (= (slew_time Star5 Star0) 36.56)
        (= (slew_time Star0 Star5) 36.56)
        (= (slew_time Star5 GroundStation1) 8.59)
        (= (slew_time GroundStation1 Star5) 8.59)
        (= (slew_time Star5 GroundStation2) 62.86)
        (= (slew_time GroundStation2 Star5) 62.86)
        (= (slew_time Star5 Phenomenon3) 10.18)
        (= (slew_time Phenomenon3 Star5) 10.18)
        (= (slew_time Star5 Phenomenon4) 64.5)
        (= (slew_time Phenomenon4 Star5) 64.5)
        (= (slew_time Phenomenon6 Star0) 77.07)
        (= (slew_time Star0 Phenomenon6) 77.07)
        (= (slew_time Phenomenon6 GroundStation1) 17.63)
        (= (slew_time GroundStation1 Phenomenon6) 17.63)
        (= (slew_time Phenomenon6 GroundStation2) 50.73)
        (= (slew_time GroundStation2 Phenomenon6) 50.73)
        (= (slew_time Phenomenon6 Phenomenon3) 14.75)
        (= (slew_time Phenomenon3 Phenomenon6) 14.75)
        (= (slew_time Phenomenon6 Phenomenon4) 2.098)
        (= (slew_time Phenomenon4 Phenomenon6) 2.098)
        (= (slew_time Phenomenon6 Star5) 29.32)
        (= (slew_time Star5 Phenomenon6) 29.32)
        (= (data-stored) 0)
```

```
        (= (fuel-used) 0))

(:goal (and     (have_image Phenomenon4 thermograph0)
                (have_image Star5 thermograph0)
                (have_image Phenomenon6 thermograph0)))

(:metric minimize (fuel-used))

)
```

Az előbbi PDDL-es probléma-leírásban külön pirossal kiemelve tüntettük fel azokat a részeket, amik nem szerepeltek az előző szakasz probléma-leírásában. Ezek egyik része tehát a kiindulási világállapotban igaz tények halmazát bővíti az új fogalmaknak, pontosabban a függvényeknek megfelelően. A másik újdonság pedig a jósági mérce.

Kezdjük az előbbi résszel – az új tényekkel! Sorra haladva láthatjuk, hogy először a „`satellite0`” műhold kezdeti üres tárkapacitása kerül megadásra (mondjuk 1000 megabájt), majd a kezdeti üzemanyag mennyisége (mondjuk 112 liter). Ezt követi a különböző célpontokról különböző üzemmódban készíthető felvételek mérete (mondjuk megabájtban), majd a különböző célpontok közti forduláshoz szükséges idő (mondjuk szekundumban)[26], végül pedig a kezdetben felhasznált összes tárkapacitás és üzemanyag mennyisége (ami nyilván zérus). – 0-argumentumú függvényt nem muszáj bezárójelezni!

FONTOS (!!!): Vegyük észre, hogy amíg a domain-leírásban (a cselekvések következmény-részében) `increase`, `assign`, … beépített eljárások által tudtunk csak értéket adni a numerikus változóknak (függvényeknek), addig a probléma-leírásban (lásd. fentebb) az ”=„ beépített eljárás (egyesítés) segítségével. Erre tehát ügyeljünk amellett, hogy – ahogy azt már egyszer kifejtettük – az ”=„ beépített eljárás sem mindig érték-adó (logikai változóknál), néhol érték-összehasonlító (numerikus változóknál).

A kezdeti tények (mind predikátumok, mind pedig predikátumok formájában megadott numerikus értékek) felsorolását a jósági mérce megadása követi. Ez jelenleg a következő:

```
(:metric minimize (fuel-used))
```

Jósági mércénk (ami – vegyük észre! – a konkrét problémához tartozik) tehát igen egyszerű: azt írja elő, hogy a leírt probléma megoldásai minimalizálják az összesen felhasznált üzemanyag mennyiségét. Ez egy egyszerű, ésszerű előírás, ami rákényszeríti a tervkészítőket arra, hogy jelen probléma megoldásakor optimalizálják az összesített üzemanyag-fogyasztást.

Természetesen nem csak minimalizálni (`minimize`) lehet, hanem maximalizálni (`maximize`) is. Nyilván ez utóbbi jelen esetben (az üzemanyag-fogyasztásra vonatkozóan) elég ésszerűtlen volna, viszont más esetekben, más jósági mérce megadása mellett még ésszerű lehet. Továbbá az optimalizálandó jósági mércének sem kell feltétlen ilyen egyszerűnek lennie, mint most – tetszőlegesen összetett numerikus formula lehet.

[26] Itt érdemes felfigyelni arra, hogy az értékek következetesen szimmetrikusnak lettek definiálva. Azaz A-ból B-be ugyanannyi ideig tart a fordulás, mint B-ből A-ba. Elfogadható (egyszerűsítő) feltevés.

Probléma-megoldás

Lássuk tehát, hogy milyen megoldást ad az LPG a fentebb definiált problémára:

```
0:   (SWITCH_ON INSTRUMENT0 SATELLITE0) [1]
0:   (TURN_TO SATELLITE0 PHENOMENON4 PHENOMENON6) [1]
1:   (TURN_TO SATELLITE0 GROUNDSTATION2 PHENOMENON4) [1]
2:   (CALIBRATE SATELLITE0 INSTRUMENT0 GROUNDSTATION2) [1]
3:   (TURN_TO SATELLITE0 PHENOMENON4 GROUNDSTATION2) [1]
4:   (TAKE_IMAGE SATELLITE0 PHENOMENON4 INSTRUMENT0 THERMOGRAPH0) [1]
5:   (TURN_TO SATELLITE0 PHENOMENON6 PHENOMENON4) [1]
6:   (TAKE_IMAGE SATELLITE0 PHENOMENON6 INSTRUMENT0 THERMOGRAPH0) [1]
7:   (TURN_TO SATELLITE0 PHENOMENON3 PHENOMENON6) [1]
8:   (TURN_TO SATELLITE0 STAR5 PHENOMENON3) [1]
9:   (TAKE_IMAGE SATELLITE0 STAR5 INSTRUMENT0 THERMOGRAPH0) [1]
```

A megoldás eltér attól, amit az előző, II.3.2-es szakaszban kaptunk. Ennek oka, hogy a tervkészítő jelen esetben az üzemanyag-fogyasztásra is tekintettel volt. Viszont legalább van megoldás. Szerencsére tehát a célként előírt felvételek megléte nem igényelt több üzemanyagot, mint amennyi minimálisan szükséges, és az adatkapacitással se volt gond (értsd. maradéktalanul elfértek). Vegyük kicsit részletesebben is szemügyre a fenti tervet!

Pirossal jeleztük azokat a lépéseket, melyek az előző szakaszban egyetlen lépésben lettek megoldva. Mi ennek az oka? Nyilván az üzemanyag-fogyasztás, ami a fordulási idővel (`slew_time`) van egyen-arányban. Tehát a probléma-leírásban érdemes szemügyre vennünk a megadott fordulási időket. A következő idevonatkozó értékek szerepelnek:

```
(= (slew_time GroundStation2 Phenomenon6) 50.73)
(= (slew_time Phenomenon4 Phenomenon6) 2.098)
(= (slew_time GroundStation2 Phenomenon4) 39.73)
```

Ezek szerint valóban ésszerű volt „`Phenomenon6`” felől először „`Phenomenon4`” felé fordulni, majd onnan „`GroundStation2`” felé, mint azonnal a „`GroundStation2`” felé, hiszen: `50.73 > 2.098 + 39.73 = 41.828`

A másik eset is hasonló:

```
(= (slew_time Star5 Phenomenon6) 29.32)
(= (slew_time Phenomenon3 Phenomenon6) 14.75)
(= (slew_time Star5 Phenomenon3) 10.18)
```

Itt tehát: `29.32 > 14.75 + 10.18 = 24.93` adódik. Ezen túl a T. Olvasóra bízzuk annak ellenőrzését, hogy vajon lett volna-e még ennél is hatékonyabb, kevesebb időt igénylő fordulás-sorozat az előírt célok megvalósítására.

II.3.4. Időzítések bevezetése

Az előbbi szakaszban láthattuk, hogy a tervkészítésben az erőforrások (pl. idő, energia, pénz, üzemanyag) igen fontos szerepet játszanak, különös tekintettel az időre. Az időnek elsődleges szerepe van, és volt eddig is. Gondoljunk csak a mutex cselekvésekre, vagy a párhuzamos végrehajtásra. Az idő az eddigi megoldásokban is igen fontos szerepet töltött be, megjelent, csak hát – az egyszerűség kedvéért – eddig alapértelmezésben minden cselekvés végrehajtási ideje egységnyi volt.

Az eddigiekben tehát igazából csak a cselekvések egy logikai sorrendezését adtuk meg (a probléma-megoldásokban) független attól, hogy mennyi időbe telik a végrehajtásuk. Ez ellentmondásnak tűnhet az előbbi szakaszban leírtakkal, hiszen ott már bevezettük az idő fogalmát. Mindazonáltal az ott bevezetett idő-fogalom a mi saját, egyénileg definiált fogalmunk volt, amiről a tervkészítő egyáltalán nem értesült.

Vegyük csak észre, hogy az előbbi szakasz végén, a probléma-megoldás részben szereplő tervben például a `(TURN_TO SATELLITE0 PHENOMENON4 PHENOMENON6)` lépés – a tervkészítő szerint – a 0. Időpillanatban kezdődik, és az 1. Időpillanatban ér végett, miközben mi tudjuk, hogy a „`PHENOMENON6`” célpont felől a „`PHENOMENON4`” célpont felé `2.098` időegységig tart a fordulás.

Ebben a szakaszban ezért valahogy megpróbáljuk a tervkészítő tudomására hozni, ha valahol valóban időről van szó. Ehhez – a PDDL2.1 óta adott lehetőségekhez mérten – bővítjük, pontosabban átszerkesztjük az előbbi PDDL-leírást:

Domain-leírás

Kezdjük a domain-leírás átírásával! Ezen belül lényegében csak a cselekvés-sémákra kell fókuszálnunk, mivel minden más – a követelmények kivételével – változatlan. Ennek megfelelően az időzítéseket is tartalmazó domain-leírás a következő:

```
(define (domain satellite)
(:requirements :strips :typing :equality :fluents :durative-actions)
(:types satellite direction instrument mode)
(:predicates
        (on_board ?i - instrument ?s - satellite)
        (supports ?i - instrument ?m - mode)
        (pointing ?s - satellite ?d - direction)
        (power_avail ?s - satellite)
        (power_on ?i - instrument)
        (calibrated ?i - instrument)
        (have_image ?d - direction ?m - mode)
        (calibration_target ?i - instrument ?d - direction))
(:functions
        (data_capacity ?s - satellite)
        (data ?d - direction ?m - mode)
        (slew_time ?a ?b - direction)
        (data-stored)
        (fuel ?s - satellite)
        (fuel-used))
```

```
(:durative-action turn_to
   :parameters (?s - satellite ?d_new ?d_prev - direction)
   :duration   (= ?duration (slew_time ?d_new ?d_prev))
   :condition  (and    (at start      (pointing ?s ?d_prev))
                       (over all      (not (= ?d_new ?d_prev)))
                       (at end        (>= (fuel ?s) (slew_time ?d_new ?d_prev))))
   :effect     (and    (at start      (not (pointing ?s ?d_prev)))
                       (at start      (decrease (fuel ?s) (slew_time ?d_new ?d_prev)))
                       (at end        (pointing ?s ?d_new))
                       (at end        (increase (fuel-used) (slew_time ?d_new ?d_prev))))
)

(:durative-action switch_on
   :parameters (?i - instrument ?s - satellite)
   :duration   (= ?duration 2)
   :condition  (and    (at start      (power_avail ?s))
                       (over all      (on_board ?i ?s)))
   :effect     (and    (at start      (not (calibrated ?i)))
                       (at start      (not (power_avail ?s)))
                       (at end        (power_on ?i)))
)

(:durative-action switch_off
   :parameters (?i - instrument ?s - satellite)
   :duration   (= ?duration 1)
   :condition  (and    (at start      (power_on ?i))
                       (over all      (on_board ?i ?s)))
   :effect     (and    (at start      (not (power_on ?i)))
                       (at end        (power_avail ?s)))
)

(:durative-action calibrate
   :parameters (?s - satellite ?i - instrument ?d - direction)
   :duration   (= ?duration 5)
   :condition  (and    (over all      (pointing ?s ?d))
                       (over all      (calibration_target ?i ?d))
                       (over all      (on_board ?i ?s))
                       (over all      (power_on ?i))
                       (at end        (power_on ?i)))
   :effect     (at end (calibrated ?i))
)

(:durative-action take_image
   :parameters (?s - satellite ?d - direction ?i - instrument ?m - mode)
   :duration   (= ?duration 7)
   :condition  (and    (at start      (>= (data_capacity ?s) (data ?d ?m)))
                       (over all      (calibrated ?i))
                       (over all      (on_board ?i ?s))
                       (over all      (supports ?i ?m) )
                       (over all      (power_on ?i))
                       (over all      (pointing ?s ?d))
                       (at end        (power_on ?i)))
   :effect     (and    (at start      (decrease (data_capacity ?s) (data ?d ?m)))
                       (at end        (have_image ?d ?m))
                       (at end        (increase (data-stored) (data ?d ?m))))
)
)
```

A leírásban pirossal emeltük ki azokat a részeket, melyek különböznek az előző, II.3.3-es szakaszban olvashatóktól. Ezen belül látható, hogy először is a követelmények bővültek egy „`:durative-actions`” flag-gel. Ezen felül pedig már csak a cselekvés-sémák leírása változott. Immár nem „`:action`”-ökről, hanem „`:durative-action`”-ökről (időzített cselekvésekről) beszélünk.

Ennek megfelelően a cselekvés-sémák leírása kiegészült egy időzítést leíró résszel: Ennek első eleme a „`:duration`", míg ez után következik az értéke, ami lényegében egy tetszőlegesen összetett numerikus formula, amely a „`?duration`" változónak ad értéket. Ez a változó tehát kulcsszó!

A fenti domain-leírásban kétféle időzítés szerepel: egyrészt egy olyan, amikor konkrét, konstans számértéket adunk időzítésül. Ez az egyszerűbb eset. Másrészt (a „`turn-to`" cselekvés esetében) olyan időzítés is szerepel, ahol már egy numerikus változó értékét adjuk meg. Az előbbi például:

```
(= ?duration 7)
```

Az utóbbi pedig:

```
(= ?duration (slew_time ?d_new ?d_prev))
```

Habár az előbbi fajtájú cselekvés mindig és mindenkor a megadott ideig fog tartani, míg az utóbbi fajta (ahol egy numerikus formula értékével történik egyesítés), a tudásbázis változásával, akár már a terv végrehajtása során is más, és más időzítéssel futhat le, mindkét fajtát egységesen ***fix-időtartamú cselekvés***nek nevezik.[27]

A cselekvések időtartamának megadását az elő- és utófeltételek megadása követi. Itt kétféle változás figyelhető meg. Egyrészt immár „`:precondition`" helyet „`:condition`" string szerepel az előfeltételeknél, másrészt az előfeltételeket és következményeket leíró logikai állítások összetétele is megváltozott.

A bennük szereplő tényeket immár egy időzítési deklarációval is „körbezártuk". Megadhatjuk (mind az előfeltételekben, mint a következményekben), hogy mikor kell igaznak lennie a ténynek: már a cselekvés végrehajtásának megkezdésekor (`at start`); a cselekvés végrehajtása során mindvégig (`over all`); vagy pedig már csak a cselekvés végrehajtásának befejeztekor (`at end`).

Természetesen egy tényt akár több helyen is feltüntethetünk (pl. az előfeltételeken belül). Ennek azért van értelme, mert az „`over all`" nyílt időintervallumot azonosít, melynek két végpontja az „`at start`" és „`at end`" által azonosított kezdet, és vég. Példának okáért vegyük a „`calibrate`" cselekvést. Ha annak előfeltételében nem „`over all`", hanem „`at start`" szerepelne a `(pointing ?s ?d)` kapcsán, akkor a kalibráláshoz elég lenne – a kezdetek kezdetén – egy pillanatra a kalibrálási cél felé fordulni, megkezdeni a kalibrálást, majd máris más irányba fordulni. Ez ésszerűtlen volna, hiszen a kalibráláshoz értelemszerűen mindvégig a kalibrálási cél felé kell nézni, míg a művelet véget nem ér.

Amíg az előfeltételek esetében a tények igazságát vizsgáljuk, addig a következményeknél nyilván a tudásbázisba szúrjuk be, vagy vesszük el onnan őket.[28] Ezért tehát a

[27] Vannak nem fix-időtartamú cselekvések is, ám mi ezekkel a jelen anyagban nem foglalkozunk.
[28] Hasonlóan a Prolog `assert`, és `retract` (beépített) eljárásaihoz..

következményekben megadott időzítésekkel igencsak csínján kell bánni. Egyrészt nyilván csakis az elején (`at start`), vagy a végén (`at end`) fejthet ki hatást a cselekvés.

Másrészt javasolt a „***konzervatív frissítés***” nevezetű metodika betartása. Ezek szerint a cselekvés következményeiben, amennyiben egy erőforrás fogyasztása (pl. `decrease`, `not`) történik, úgy azt a kezdetek kezdetén (`at start`) célszerű végrehajtani, míg ha a cselekvés hatására erőforrás termelődik (pl. ponált tény szúródik be, `assign`, `increase`), úgy azt a végén (`at end`) célszerű foganatosítani. A módszer következetes használatával – ha meggondoljuk, akár párhuzamos végrehajtás során is – biztosított lesz, hogy ne emésszünk fel erőforrásokat még azelőtt, mielőtt megtermelődnének.

Az előfeltételekben és következményekben szereplő időzítések ilyetén felosztása végső soron diszkrét, mivel kizárólag három időintervallumot különböztetünk meg: *kezdet* (`at start`), *közben* (`over all`), és *vég* (`at end`). Ezért nevezik a fenti domain-leírásban szereplő cselekvéseket ***fix-időtartalmú diszkrét cselekvések***nek.[29]

A cselekvések időtartamának megadásában szerepelt egy kiemelt „`?duration`” változó. Felmerülhet a kérdés, hogy: *Vajon a cselekvés bemenő-paramétereihez hasonlóan szerepeltethetjük-e ezt is az előfeltételekben, vagy a következményekben?*

A válasz: *részben*. Az előfeltételekben nyilván nem szerepeltethetjük, mivel általánosságban akkor – a cselekvés megkezdése előtt – még nem tisztázott, hogy mennyi ideig fog tartani a tényleges végrehajtás (a „`?duration`” változó még nem behelyettesített). Viszont a következményekben már bátran szerepeltethetjük, sőt, néha kifejezetten szükséges, hogy az időtartamtól függően változtassuk a tudásbázis tartalmát.

Probléma-leírás

Az előbbiekben részletesen áttekintettük, hogy milyen változtatásokkal jár az időzítések bevezetése a domain-leírásra nézve. Láthattuk, hogy szinte az egész domain-leírást (melynek túlnyomó részét a cselekvés-sémák teszik ki) át kellett írnunk. Szerencsére azonban a probléma-leírással kapcsolatban már semmi dolgunk sincs. Lényegében tehát az előző, II.3.3-as szakaszban adott probléma-leírás egy-az-egyben megfelel most is.

Mindazonáltal, az érdekesség kedvéért, kihasználva az időzítés adta lehetőségeket, írjuk át az előbbi jósági mércét a következőre:

```
(:metric minimize (total-time))
```

Minden más tehát maradjon a régiben, csak és kizárólag a jósági mércét írjuk át az előbbire. A „`total-time`” itt a megoldás (terv) által igényelt összes időt jelöli, és beépített, előre definiált eljárásnak tekinthető ellenben mondjuk a „`fuel-used`”, vagy a „`data-stored`” függvényekkel.

[29] Természetesen folytonos (hatású) cselekvések is vannak, ám ezeket – hasonlóan a feltételes (hatású) cselekvésekhez – a jelen anyagban nem tárgyaljuk.

Probléma-megoldás

Az előbbi PDDL reprezentációval adott probléma megoldásakor feltűnhet, hogy a tervkészítő nagyságrendekkel hamarabb adott megoldást, mint előbb (lásd. II.3.3). Viszont a kapott terv gyakorlatilag most is ugyanaz:

```
0.0003:   (SWITCH_ON INSTRUMENT0 SATELLITE0) [2.0000]
0.0005:   (TURN_TO SATELLITE0 PHENOMENON4 PHENOMENON6) [2.0980]
2.0988:   (TURN_TO SATELLITE0 GROUNDSTATION2 PHENOMENON4) [39.7300]
41.8290:   (CALIBRATE SATELLITE0 INSTRUMENT0 GROUNDSTATION2) [5.0000]
46.8293:   (TURN_TO SATELLITE0 PHENOMENON4 GROUNDSTATION2) [39.7300]
86.5595:   (TAKE_IMAGE SATELLITE0 PHENOMENON4 INSTRUMENT0 THERMOGRAPH0) [7.0000]
93.5598:   (TURN_TO SATELLITE0 PHENOMENON6 PHENOMENON4) [2.0980]
95.6580:   (TAKE_IMAGE SATELLITE0 PHENOMENON6 INSTRUMENT0 THERMOGRAPH0) [7.0000]
102.6582:   (TURN_TO SATELLITE0 PHENOMENON3 PHENOMENON6) [14.7500]
117.4085:   (TURN_TO SATELLITE0 STAR5 PHENOMENON3) [10.1800]
127.5888:   (TAKE_IMAGE SATELLITE0 STAR5 INSTRUMENT0 THERMOGRAPH0) [7.0000]
```

Az időzítésektől eltekintve tehát a kapott megoldás most is ugyanaz, mint előbb. Ennek oka, hogy amíg az előbbiekben az összesen elfogyasztott üzemanyag mennyiségét próbáltuk minimalizálni (miközben az üzemanyag-fogyasztás 1:1 arányban volt a fordulási idővel), addig most magát az összidőt. A jelen felállásban (probléma-leírásban) tehát akkor lenne baj, ha az előbbi szakaszban kapott optimális terv nem egyezne meg a mostanival. Habár példánkban csupán csak egyetlen műhold-objektum szerepel. Több műhold esetén az egyes műholdak üzemanyag fogyasztása összeadódna, míg az általuk végzett műveletek végrehajtása időben átlapolódhatna. Ilyen értelemben tehát a mostani jósági mérce mégis csak más, mint előbb.

II.3.5. Származtatott predikátumok

A következőkben olyan elemeket vezetünk be a mintapélda kidolgozásába, melyek a PDDL2.2-es változatában (azaz 2004-ben) jelentek meg először: foglalkozunk a származtatott predikátumokkal, és a következő szakaszban az időzített kezdeti literálokat is megismerjük. Mindkét kiegészítés érdekes, és fontos, ám végső soron nem annyira alapvető, mint az előzőekben bemutatott feature-ök. Az előzőekben ugyanis – vegyük észre – rendre a PDDL három szintjét vettük végig: (1) *STRIPS-es leírás*, (2) *numerikus leírás*, és (3) *temporális leírás*. Most tehát vizsgáljuk meg ezek egy-két kiegészítését.

A ***származtatott predikátumok***at az különbözteti meg az eddig megismert predikátumoktól, hogy igazságukat – a visszafelé láncolt szabály-alapú szakértői rendszerekhez hasonlóan – következtetési szabályok alapján dönthetjük el. A származtatott predikátumok tehát egy-egy megfelelő „`:derived`” szabály következmény-részében foglalnak helyet, aminek előfeltételében elemi, vagy származtatott predikátumok tetszőleges logikai kombinációja lehet[30]. Amennyiben az így megadott feltétel-rész igaz, úgy igaz a megfelelő származtatott predikátum.

[30] Elvben **numerikus változók** értékvizsgálata is helyet foglalhatna **a származtatott predikátumok feltétel részében**, azonban az LPG jelen megvalósítása ezt sajnos még nem támogatja. Hasonlóan nem javallott **időzített kezdeti literálok** feltétel-részben történő szerepeltetése, ugyanis – mivel nem volt része a 2004-es IPC versenynek – az **LPG nincs** rá **felkészítve**.

Mivel tehát a származtatott predikátumok definíciójában származtatott predikátumok is szerepelhetnek, ezért lehetőségünk van önhivatkozó definíciók kialakítására, pl.:

```
(:derived   (above ?a ?c - cube)
            (or   (on ?a ?c)
                  (exists (?b - cube) (and      (on ?a ?b)
                                                (above ?b ?c))))
)
```

Ezek szerint tehát az „`?a`” kocka a „`?c`” kocka *fölött van* (`above`) akkor, ha vagy „`?a`” *rajta van* (`on`) „`?c`”-n, vagy létezik olyan „`?b`” kocka, amelyiken „`?a`” rajta van, miközben „`?b`” a „`?c`” fölött van. Ezzel tehát egyfajta ***tranzitív kapcsolat***ot definiáltunk a kockák között. De természetesen mindenféle önhivatkozás nélkül, csupán csak elemi tények „összevonására” is használhatjuk a származtatott predikátumokat.

Az egyszerűség kedvéért[31] ebben a szakaszban csakis ez utóbbi esettel foglalkozunk. Térjünk is vissza eddigi mintapéldánkhoz, és vegyük szemügyre a műszerek ki/bekapcsolásáért felelős cselekvések (`switch_on`, és `switch_off`) környezetét.

Mint tudjuk egyelőre egy műholdon egyszerre csak egy műszer lehet bekapcsolva. Ezt biztosítja a „`power_avail/1`” (szemafor-jellegű) predikátum. A bekapcsolás eredménye ebben az esetben az, hogy a műszer nyomban áram alá kerül (`power_on`), miközben a „`power_avail`” hamissá válik az adott műholdra nézve.

Ez azonban így, ebben a formában nyilván nem igaz. A „`power_avail`” predikátum jelentésének értelmezése kétes. A predikátum nem a műhold energia-ellátásának meglétére vonatkozik (ami megszűnik, ha valamit bekapcsolunk), hanem inkább arra, hogy van-e még elegendő energia egy további műszer bekapcsolásához. Valójában azonban a helyzet a következő:

Egy műszer csak akkor kapcsolható be egy műholdon, ha a műhold fedélzetén van, és ki van kapcsolva. A bekapcsolás hatására csak akkor kerül áram alá, ha más műszer nincs bekapcsolva a műholdon (mert például, egyébként túlterhelődik a hálózat).

Ez így elég természetes. Most azonban nem így van! A műszer bekapcsolásának feltétele, hogy „`power_avail`” igaz legyen. Pedig hát végső soron – ha meggondoljuk – bármikor bebillenthetnénk azt a „bekapcsolás gombot” a műszeren, csak legfeljebb nem kerülne áram alá, nem indulna be. Ráadásul ennek számtalan egyéb oka is lehet azon felül, hogy van-e még más műszer is bekapcsolva a fedélzeten. Elképzelhető, hogy a műszer meghibásodik, vagy a

[31] …és mivel az LPG jelen verziójában van egy-két bizonytalanság az önhivatkozó származtatott predikátumokat előfeltételeikben tartalmazó cselekvések mutex-jellegének megállapításakor. A PDDL2.1-ben ugyanis két (teljesen behelyettesített) cselekvés „interferált”, ha az egyik olyan – ponált vagy negált – tényt tartalmazott következményei közt, amely szerepelt a másik előfeltételeiben. Most, a PDDL2.2-ben a helyzet annyival bonyolultabb, hogy az interferencia már akkor is fellép, ha az egyik cselekvés olyan tényt tartalmaz következményei közt, melyből *következik* a másik valamely (származtatott) előfeltétele.

csatlakozás mond csődöt, stb. A későbbiekben már ezeket a részleteket is be kellhet vinnünk az eddigi leírásba.[32]

Ezért ezt a "domain-specifikus logikát" – pl. a "`(power_on ?i)`" tény igazságának eldöntéséhez szükséges speciális logikai struktúrát – célszerű különválasztani a cselekvésektől. Főként erre valók a származtatott predikátumok, mint logikai szabályok. A cselekvések előfeltételeiből kiemelhetjük a megfelelő részeket, és modulárisan, struktúráltan áthelyezhetjük ezekbe a logikai szabályokba.

Domain-leírás

Az előbbi érvelésnek megfelelően alakítsuk át az eddigi domain-leírást úgy, hogy a „`power on/1`" tények igazsága immár származtatott legyen! Vezessük be a következőt:

```
(:derived (power_on ?i - instrument)
          (and (switched_on ?i)
               (not (exists (?j - instrument ?s - satellite) (and  (on_board ?i ?s)
                                                                   (on_board ?j ?s)
                                                                   (not (= ?i ?j))
                                                                   (switched_on ?j)))))
)
```

Mindemellett azonban a domain-leírás „`:predicates`" részében továbbra is szerepltetnünk kell a „`power_on/1`" predikátumot.[33] Ráadásul a leírás „`:requirements`" részét is ki kell egészítenünk egy „`:derived-predicates`" flag-gel! A „`power_avail/1`" predikátum helyére pedig vezessünk be egy gyakorlatibb, „`switched_on/1`" predikátumot, amely akkor igaz, ha a műszer be van kapcsolva.

Ezen felül – az „működés" eldöntéséhez szükséges logika kiemelésének köszönhetően – egyszerűsítsük le a „`switch_on`", és „`switch_off`" cselekvéseket a következőképp:

```
(:durative-action switch_on
   :parameters    (?i - instrument ?s - satellite)
   :duration      (= ?duration 2)
   :condition     (and  (at start   (not (switched_on ?i)))
                        (over all   (on_board ?i ?s)))
   :effect        (and  (at start   (not (calibrated ?i)))
                        (at end     (switched_on ?i)))
)

(:durative-action switch_off
   :parameters    (?i - instrument ?s - satellite)
   :duration      (= ?duration 1)
   :condition     (and  (at start   (switched_on ?i))
                        (over all   (on_board ?i ?s)))
   :effect        (and  (at start   (not (switched_on ?i))))
)
```

[32] A PDDL leplezetlen célkitűzése a kezdetektől fogva, hogy meg tudja ragadni a tervkészítési környezet „*fizikáját*". Ezért is célszerű redukálni az absztrakt (pl. „`power_avail`" jellegű) predikátumok számát…
[33] Elhanyagolható redundancia. Az anyag keretein túlmutató PDDL-tervezői döntések állnak mögötte.

Látható, hogy immár nincs megkötve, hogy pl. a „`switch_on`” cselekvés csak akkor hajtható végre, ha más még nincs bekapcsolva. Nem. Immár bármikor bármelyik műszert be lehet kapcsolni, csak legfeljebb nem fognak működni (`power_on`).

A leírás többi része változatlan az előző, II.3.4-es szakaszban foglaltakhoz képest. Esetleg még csak arra hívnám fel a figyelmet (FONTOS!!!), hogy vegyük észre: származtatott predikátumok csak a cselekvések előfeltételei közt szerepelnek, a következmények közt soha. Ennek oka, hogy a származtatott predikátumok igazsága végső soron így is, úgy is adódik a megfelelő elemi tények igazságából, így felesleges még ezzel is komplikálni a leírást, nehezíteni a tervkészítők dolgát.

Probléma-leírás

A probléma-leírás gyakorlatilag ugyancsak változatlan az előző szakaszban foglaltakhoz képest azzal a csekély, de nem elhanyagolható különbséggel, hogy – mivel immár nem beszélünk „`power_avail/1`” predikátumról – a kezdeti tények közül kikerül a következő:

```
(power_avail satellite0)
```

A származtatott predikátum bevezetésének köszönhetően tehát itt is egyszerűsödött a helyzet.

Probléma-megoldás

A megoldás azonban – nem meglepő módon – továbbra is változatlan. Viszont – érdemes észrevennünk – a.származtatott predikátum bevezetése itt is láthatóan jótékony hatással volt: a tervkészítés folyamata újfent többszörösére gyorsult.[34]

II.3.6. Időzített kezdeti literálok

Eddig a PDDL probléma-leírásban csak azt adtuk meg, hogy a kezdeti (nulladik) időpillanatban mely tények igazak. Ezért a tudásbázis csak és kizárólag a cselekvések hatására változhatott (bővülhetett, vagy szűkülhetett). Most azonban bevezetjük annak a lehetőségét, hogy a tudásbázis előre meghatározott, determinisztikus módon, előre megadott időpillanatokban bővüljön, vagy szűküljön.

Tetszőleges elemi literál (behelyettesített argumentumokkal rendelkező elemi predikátum – tény) igazsága befolyásolható ilyen előre időzített exogén környezeti események formájában. Mielőtt azonban rátérünk a részletekre, a példa kedvéért tegyük fel, hogy a probléma informális leírása a következőkkel bővült:

A célpontokról csak akkor készíthető felvétel, ha az adott műhold felől láthatóak. Nem elég csupán a célpont irányába nézni.

[34] Úgy látszik, nem csak az emberi munkavégzést gyorsítja meg, ha „áttekinthetőbb” a probléma.

Domain-leírás

Az előbb megfogalmazott informális elvárás elég ésszerűen hangzik – még akár meglepő is lehet, hogy eddig nem vettük figyelembe. Ennek azonban oka van. Gondoljunk csak bele: a célpontok láthatóságát – a műhold és a célpont térbeli elhelyezkedése mellett – a különböző égi objektumok röppályája, és egyéb, előre nem feltétlen látható események befolyásolják. Eleddig nem tudtunk exogén eseményeket (melyek „rendszeren kívülről" jönnek, s amikre így „rendszeren belül" nincs magyarázat) ábrázolni. Mindent meg kellett indokolnunk; szabályszerű, dedukálható módon le kellett írnunk.

Ebben az esetben azonban kezelhetetlen bonyolultsággal állunk szemben. Nem vagyunk képesek szabályszerű módon leírni azt a mechanizmust, amely a célpontok – egyes műholdak felől vett – láthatóságát befolyásolja. Túlontúl összetett ez a rendszer, még ha nem is feltétlen véletlenszerű. Célszerű tehát rendszeren kívülinek, exogénnak titulálni az eseményt (amely a láthatóságot befolyásolja), és csupán csak bekövetkezését vizsgálni.[35] Vezessünk tehát be egy predikátumot, amellyel adott célpont adott műhold felől vett láthatóságát reprezentáljuk:

```
(visible ?s - satellite ?d - direction)
```

Immár tehát ezt a predikátumot is szerepeltessük a domain-leírás „`:predicates`" részében! Továbbá a követelmények között szerepeltessük a „`:timed-initial-literals`" flag-et! Ezen felül a „`take_image`" cselekvés előfeltételei közé értelemszerűen ékeljük be a következő tényt:

```
(over all   (visible ?s ?d))
```

Más kiegészítésre már nincs szükség az előző, II.3.5-ös szakaszban látott domain-leíráshoz képest.

Probléma-leírás

Az ***időzített kezdeti literálok*** láthatóan nem képezték a domain-leírás részét. Ennek oka, hogy amíg a származtatott-jelleg a predikátumok sajátja volt, addig az időzítettség már nem az.[36] Bármely behelyettesített argumentumú *elemi* predikátum (*elemi* literál) lehet időzített. Ezért végső soron a domain-leírásban sem kellett volna semmin se változtatnunk (a követelmény-rész kiegészítésétől eltekintve), hacsak nem vezetünk be valami újat, mint most a „`visible/2`" predikátumot.

[35] …hasonlóan a valószínűségszámításhoz, amely nem konstruktív olyan értelemben, hogy az egyes események bekövetkezéséhez egy-egy valószínűséget rendel csupán, és *nem magyarázza meg*, hogy az esemény valójában miért is következik be (ha bekövetkezik). A környezet tehát lehet determinisztikus, ha az ágens számára nem hozzáférhető, akkor véletlenszerűségekkel kell(het) számolnia. Az ágens számára tehát ekkor a környezet *nem-determinisztikusnak tűnhet*. Ezért is indokolt a „hozzáférhetőség", és a „determinizmus" fogalmainak különválasztása. Szerencsére azonban mi ebben az anyagban kizárólag determinisztikus tervkészítéssel foglalkozunk, ahol kizárólag teljesen hozzáférhető környezetekről esik szó.

[36] …hasonlóan ahhoz (lásd. előző lábjegyzet) , ahogy a valószínűség sem „a dolgok sajátja", hanem végső soron *szubjektív*, feltéve, hogy a világ valóban determinált, csak túlontúl komplex ahhoz, hogy kauzálisan/analitikusan le tudjuk írni, és ezért folyamodunk a valószínűségszámítás eszköztárához.

Ennek megfelelően – a probléma-leírásban – a kezdetben igaz tényeket vonatkozó „`:init`” részt a következő tényekkel egészíthetjük ki:

```
(visible satellite0 GroundStation1)
(visible satellite0 GroundStation2)
(visible satellite0 Phenomenon3)
(at 7 (visible satellite0 Star5))
(at 70 (visible satellite0 Phenomenon6))
(at 700 (visible satellite0 Phenomenon4))
(at 1000 (not (visible satellite0 Phenomenon3)))
(at 1007 (not (visible satellite0 Star5)))
(at 1070 (not (visible satellite0 Phenomenon6)))
(at 1700 (not (visible satellite0 Phenomenon4)))
```

Ezekből az első három már a 0. időpillanattól kezdve igaz, míg a második három literál csak rendre a 7., 70., és 700. időpillanattól lesz igaz tény – addig hamis[37]. Ráadásul mindez csak rendre az 1007., 1070., és 1700. időpillanatig tart, amikor is a „`Star5`”, „`Phenomenon6`”, és „`Phenomenon4`” láthatósága megszűnik (mert például, újra eléjük kerül egy égitest). Előtte pedig, az 1000. időpillanatban már a „`Phenomenon3`” is elvész a „`satellite0`” műhold szeme elöl…

Az időzített kezdeti literálok szintaxisa, és szemantikája tehát igen egyszerű. Legtöbbször – a fentihez hasonló módon – ***időablakok*** definiálására használják őket (amelyen belül nyitva tart a bolt; az emberek dolgoznak; lassú a forgalom; süt a nap; foglalt a tanterem; senki sem válaszol leveleinkre, mert mind órán vannak, stb).

Probléma-megoldás

Az LPG viszonylag csekély lassulás után a következő megoldást adta a fenti problémára:

```
0.0003:   (SWITCH_ON INSTRUMENT0 SATELLITE0) [2.0000]
0.0005:   (TURN_TO SATELLITE0 PHENOMENON4 PHENOMENON6) [2.0980]
2.0988:   (TURN_TO SATELLITE0 GROUNDSTATION2 PHENOMENON4) [39.7300]
41.8290:   (CALIBRATE SATELLITE0 INSTRUMENT0 GROUNDSTATION2) [5.0000]
46.8293:   (TURN_TO SATELLITE0 PHENOMENON4 GROUNDSTATION2) [39.7300]
700.0015:   (TAKE_IMAGE SATELLITE0 PHENOMENON4 INSTRUMENT0 THERMOGRAPH0) [7.0000]
707.0018:   (TURN_TO SATELLITE0 PHENOMENON6 PHENOMENON4) [2.0980]
709.1000:   (TAKE_IMAGE SATELLITE0 PHENOMENON6 INSTRUMENT0 THERMOGRAPH0) [7.0000]
716.1002:   (TURN_TO SATELLITE0 PHENOMENON3 PHENOMENON6) [14.7500]
730.8505:   (TURN_TO SATELLITE0 STAR5 PHENOMENON3) [10.1800]
741.0308:   (TAKE_IMAGE SATELLITE0 STAR5 INSTRUMENT0 THERMOGRAPH0) [7.0000]
```

A kapott terven látszik, hogy a tervkészítő alkalmazkodott az időzített kezdeti literálokhoz (lásd. piros lépés), viszont korántsem optimális az idő minimalizálásának tekintetében (lásd. a probléma-leírásban szereplő metrikát). Jól láthatóan a lépések sorozata (ha más időzítéssel is) ugyanaz, mint eddig. Pedig nyilvánvalóan hamarabb végeznénk, ha először nem a „`Phenomenon4`” célpontról próbálnánk felvételt készíteni, hanem valamely – célállapot által meghatározott – más célpontról, hiszen azok jóval hamarabb válnak láthatóvá, mint a „`Phenomenon4`”…[38]

[37] Élve a „zárt világ” feltételezéssel.

[38] Vélhetően tehát ezért se lett az LPG a 2004-ben rendezett IPC verseny legjobb tervkészítője ellenben a 2002-es győzelemmel (ahol még nem voltak se időzített kezdeti literálok, se származtatott predikátumok). Mindazonáltal még így is az LPG az egyik legjobb freeware PDDL2.2 kompatibilis tervkészítő, nem is beszélve a számos elismerésről, és díjról, amit a 2004-es IPC-n kapott…

II.4. Laborgyakorlat infrastruktúrája

A következőkben a laborgyakorlat szoftveres és hardveres infrastruktúrájáról ejtünk szót.

A laborban a hallgatók virtuális gépekkel dolgoznak. A fizikai host gépeken Linux fut. Ha ezeken megfelelő felhasználóval és jelszóval bejelentkeznek, akkor elindul a laborhoz tartozó Windows-os virtuális gép, ahol a Windows-ba hallgatóként belépve kezdhetik meg a munkát.

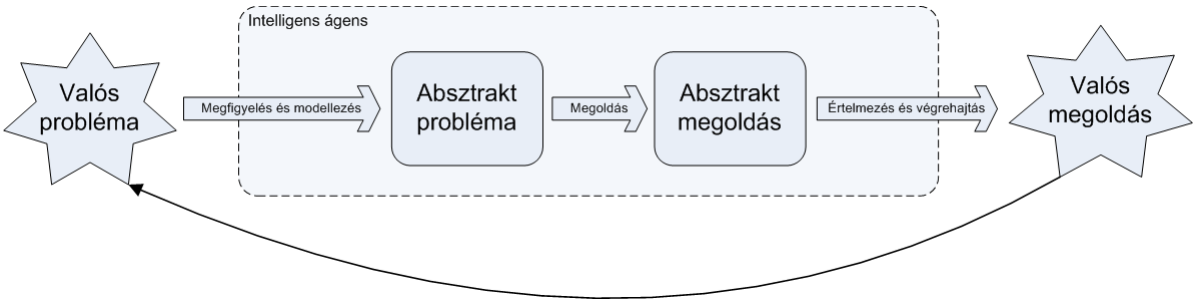
A jelen gyakorlat során lényegében a következő főbb szoftverek használata szükséges:

- **LPG** (**L**ocal search for **P**lanning **G**raphs)
- **JADE** (**J**ava **A**gent **DE**velopment framework)
- **Eclipse IDE**

A gyakorlat menete a kapcsolódó feladatok alapján nagy vonalakban a következő:

1. Kísérletezés az LPG használatával.
2. Ismerkedés néhány egyszerűbb webáruházzal.
3. A webáruházak PDDL-ben történő modellezése, és *absztrakt* vásárlási terv készítése.
4. Az absztrakt vásárlási terv adott JADE-es ágens általi *valós* végrehajtatása.

Tehát a laborgyakorlat célja, hogy a hallgatók egy bizonyos valós problémát (web-es vásárlást) képesek legyenek lemodellezni, és a modell alapján egy alkalmas tervkészítővel olyan megközelítőleg optimális megoldási tervet generálni, amely értelmesen átültethető a gyakorlatba, azaz valóban működik, és a – modellhez illő – várt eredményeket adja.[39]



II-4. ábra: a gyakorlat során a „részben emberi, részben gépi” intelligens ágensek számára adott feladat sémája

A gyakorlat során tehát a webáruházak megfigyelése és modellezése során egy PDDL leírást hozunk létre. Ennek a leírásnak 2 része van: domain- és probléma-leírás. A probléma-leírásban meg kell, hogy jelenjenek a problémában lehetséges objektumok, és kezdetben igaz tények. Ezeknek a létrehozásához a gyakorlat során biztosítunk egy egyszerű segédeszközt.

[39] Neumann szerint „*modell alatt olyan matematikai struktúra értendő, amely bizonyos informális értelmezéssel kiegészítve leírja megfigyeléseinket*”. Einstein szerint viszont „*megfigyeléseinket modelljeink tükrében értelmezzük*”… Szerencsére azonban Neumann még hozzátette, hogy: „*egy ilyen matematikai struktúra létjogosultságát csak és kizárólag az adja, hogy várhatóan működik*”. …no de vajon a helyes működésről hogyan szerzünk tanúbizonyságot? …lehet, hogy mégis modellek modelljében gondolkodunk?

A webáruház összefoglaló weboldalán **CSV** (**C**omma **S**eparated **V**alues) fájlok formájában adottak a problémában lehetséges objektumok, és kezdetben igaz tények megfogalmazásához szükséges adatok. Ahhoz, hogy ezeket könnyen és rugalmasan át tudjunk konvertálni például PDDL szintaxisúvá, a gyakorlat során az `msclab01.planning_lab.CSVtable` Java-osztályt célszerű segítségül hívnunk.

Ennek az osztálynak van egy olyan statikus metódusa (`convert`), amelyet a `main` metódus – az osztály megfelelő bemenő argumentumokkal történő meghívása esetén – arra használ, hogy felolvasson egy adott elérésen lévő CSV fájlt, és abból adott formájú szövegfájlt generáljon. Ezek szerint a `CSVtable` osztálynak legalább 4 argumentuma van:

1. *Átkonvertálandó CSV fájl elérése*
2. *Kimeneti szövegfájl elérése*
3. *A CSV fájlból kiemelendő oszlopok indexei tetszőleges sorrendben*
4. *A kimenetben a kiemelt oszlopok elemei köré/közé írandó szövegrészek listája*

A 4. bemenet valójában bemenő argumentumok tetszőleges hosszúságú listája (ideális esetben a 3. bemenetben szereplő, szóközökkel elválasztott számlista hossza+1).

Például legyen adott egy `\jade\src\msclab01\planning_lab\csv\data.csv` fájl, aminek a tartalma a következő:

```
shop;category;prodname;proddesc;prodinfo;prodprice;prodnum;prodreliability;url
blue;ram;aaa;aaa-desc ;1024;5895;2;100;http://localhost/webshops/shop1?addtocart=7501
green;hdd;bbb;bbb-desc;1536;26665;1;100;http://localhost/webshops/shop2?addtocart=7751
```

Látható, hogy ez a CSV egy olyan táblázatnak felel meg, aminek 8 oszlopa van, és a fejlécen túl mindössze 2 bejegyzése. Táblázatos formában ez a következő:

shop	category	prodname	proddesc	prodinfo	prodprice	prodnum	prodreliability	url
blue	ram	aaa	aaa-desc	1024	5895	2	100	http://localhost/webshops/shop1?addtocart=7501
green	hdd	bbb	bbb-desc	1536	26665	1	100	http://localhost/webshops/shop2?addtocart=7751

A táblázat bejegyzései tehát termékekre vonatkoznak: **(1)** áruház egyedi azonosítója, ahol az adott termék kapható; **(2)** termék típusa/kategóriája; **(3)** termék egyedi megnevezése[40]; **(4)** termék szabad szöveges leírása; **(5)** valamiféle meta-információ a termékről (pl. CPU esetében a sebesség MHz-ben; RAM és videokártya esetén a kapacitás MB-ban; HDD esetén a kapacitás GB-ban; képernyők esetén az átmérő col-ban; DVD olvasó és alaplap esetén pedig nincs értelmezve, és ezért mindig zérus); **(6)** a termék bruttó ára az adott üzleten Ft-ban; **(7)** az üzletben aktuálisan megvehető termékek darabszáma; **(8)** a termék megbízhatósága (ami arányos azzal, hogy mennyire használt); és **(9)** a termék adott üzletben történő kosárbatételéhez szükséges URL.

Tegyük fel, hogy ebből a `data.csv` fájlból szeretnénk most a `CSVtable` osztály segítségével egy olyan szövegfájlt készíteni, amelyben pl. egy PDDL-es `price/2` numerikus változónak a

[40] Elvben lehetséges volna, hogy adott fizikai termék más üzletben más egyedi néven szerepeljen, azonban ettől az egyszerűség kedvéért most eltekintünk (még hogyha a termékszótár esetében fel is készülünk rá).

kezdeti értékeit adjuk meg tényekként. A numerikus változó első argumentuma legyen a termék, a második pedig az üzlet megnevezése, ahol kapható.

Ha 0-tól 8-ig számozzuk a CSV táblázat oszlopait, akkor a 2-es, 0-ás, és 5-ös számú oszlopok értékeire lenne rendre szükségünk úgy, hogy ezek megfelelő szintaxisban legyenek adottak. A következő kimenetre lenne szükségünk:

```
(= (price aaa blue) 5895)
(= (price bbb green) 26665)
```

Ehhez az `msclab01.planning_lab.CSVtable` osztályt a következő bemenő argumentumokkal kell meghívnunk:

1. `\jade\src\msclab01\planning_lab\csv\data.csv`
2. `\jade\src\msclab01\planning_lab\csv\problem_price_data.pddl`
3. `"2 0 5"`
4. `"(= (price "`
5. `" "`
6. `") "`
7. `")"`

Ezt a hívást elvégezhetjük parancssorban, pl. a `\jade` könyvtárban a következőképp:

```
java -classpath src;lib\opencsv-2.0.jar msclab01.planning_lab.CSVtable
\jade\src\msclab01\planning_lab\csv\data.csv
\jade\src\msclab01\planning_lab\csv\problem_price_data.pddl "2 0 5" "(= (price " " " ") " ")"
```

De ugyanígy akár Eclipse-ben, egyszerűbben is végrehajtható ez a művelet, ha a *Package Explorer*-ben jobb egérgombbal rákattintunk az `msclab01.planning_lab.CSVtable` osztály forrására, és a *Run As/Open Run Dialog* opciót választva az *Arguments* fülben a következőt írjuk a *Program arguments* szövegdobozba:

```
\jade\src\msclab01\planning_lab\csv\data.csv
\jade\src\msclab01\planning_lab\csv\problem_price_data.pddl "2 0 5" "(= (price " " " ") " ")"
```

Az első 3 argumentum egyértelmű: bemenet, kimenet, és oszlopok sorrendje. Ami ezek után jön, az a töltelék az oszlopelemek között a kimenetben. A végeredmény úgy áll elő, hogy először jön egy `"(= (price "` string, aztán a 2. oszlop tartalma, aztán egy szóköz (`" "`), aztán a 0. oszlop tartalma, aztán egy záró zárójel és egy szóköz (`") "`), aztán az 5. oszlop tartalma, majd végül egy záró zárójel (`")"`). Tehát töltelék, tartalom, töltelék, tartalom, töltelék, tartalom, töltelék, … formában állíthatjuk elő a PDDL szintaxisú fájlainkat (pontosabban a probléma-leírás objektum és ténylistáját). Nyilván több különböző hívásra lesz szükségünk ahhoz, hogy minden számunka modellünkben szükséges tényt elő tudjunk állítani.[41]

Amennyiben sikerült mindent előállítanunk (pl. termékek lelőhelye, ára, kompatibilitása, és egyéb ismérvei), akkor kigenerálhatjuk a megoldási tervet LPG-vel, és ezt végrehajtathatjuk az `msclab01.planning_lab.PlanExecutorAgent` ágenssel.

[41] De ez még mindig jobb, mint hogyha manuálisan kellene begépelnünk, vagy esetleg Excel-t és Wordpad-del kombinálva Search and Replace-szel „voodooznunk”… ☺

A `PlanExecutorAgent` ágensnek csupán egyetlen paramétere van: a végrehajtandó terv elérése. Ezen felül minden mást az ágens kódjában, és környezetében kell beállítanunk.

Először is vegyük szemügyre az ágens környezetét. Az ágens lényegében 3 CSV állományra támaszkodik: **(1)** a fentihez hasonló katalógusra, ahol adott, hogy mi-hol-mennyiért kapható, és milyen URL-en tehetjük a kosárba; **(2)** áruház-szótár; és **(3)** termék-szótár.

Az **áruház szótár**ra azért van/lehet szükség, mert nem biztos, hogy a PDDL modellben ugyanaz lesz a neve egy-egy áruháznak, mint ami pl. az (1)-es CSV fájlban. Tehát itt ebben a szótárban ilyen PDDL objektum neveket társítunk össze (1)-es CSV fájlbeli áruháznevekkel. Egy PDDL objektum névhez nyilván egy, és csak egy fordítás tartozhat (lásd. pl. `\jade\src\msclab01\planning_lab\csv\shopdict.csv`).

A **termék szótár** újfent PDDL objektum neveket fordít (1)-es CSV fájlbeli terméknevekké úgy, hogy közben még egy áruháznév fordítás is szükséges. Ez azért van így, mert elvben előfordulhatna (habár a jelen gyakorlat adataiból az egyszerűség kedvéért az ilyesmit kiszűrtük), hogy ugyanaz a termék (amit ugyanúgy nevezünk a PDDL modellben) más-más néven fut a különböző áruházakban (lásd. pl. `\jade\src\msclab01\planning_lab\csv\proddict.csv`).

Ezen szótárak tartalmát az (1)-es katalógus CSV-ből kiindulva könnyedén előállíthatjuk pl. Excel-ben.

Ezen szótárak elérése szerepel a `PlanExecutorAgent` ágens kódjának elején a `DATA`, `SHOPDICT` és `PRODDICT` String-konstansokban (de nem érdemes rajtuk változtatgatni). Sokkal érdekesebb a `PlanExecutorAgent` ágens `interpretAction` metódusa! Itt kell ugyanis beállítanunk, hogy a paraméterként megadott LPG-s megoldási tervből kiolvasott cselekvéseket hogyan alakítsuk át URL-lé.

A kosárba tevős URL-eket nyilván a `\jade\src\msclab01\planning_lab\csv\data.csv` fájlból kellene kibányásznunk. Ehhez az adott megoldási tervben szereplő cselekvés nevéből és paramétereiből kell kiindulnunk. Például tegyük fel, hogy a következő, `\jade\src\msclab01\planning_lab\Planner\testplan.SOL` tervet szeretnénk végrehajtatni a `PlanExecutorAgent` ágenssel:

```
0:   (TO-CART 15TBSamsungHD154UISATAII32MBcachewinchesterECOgreen ZOLD)
1:   (CHECK-OUT ZOLD)
2:   (TO-CART 1GB800MHzDDR2RAMGeilValueSPD5 KEK)
```

Az `interpretAction` metódus egyenként kapja meg a fentebb soronként olvasható cselekvéseket String-ekből álló Vector-ok formájában. A Vector-ok 0. eleme nyilván akkor tehát a cselekvések neve – e szerint szűrhetünk az `interpretAction` metódusban. Ezért is van benne az elején az `if-then-else-if-then-…` vezérlési struktúra.

A kosárba tevős cselekvések esetén (ahol pl. a cselekvés neve `to-cart`), ott először is megpróbáljuk lefordítani a 2-es számú elemét a cselekvést reprezentáló String-Vector-nak, ami az áruházat azonosítja:

```
int[] inputidx1  = {0};
String[] input1  = {action.elementAt(2)};
outputidx        = 1;
String csvShop   = shopdict.translate(inputidx1, input1, outputidx);
```

Itt a hangsúly a `CSVtable.translate` metódus meghívásán van. A metódusnak 3 argumentuma van: **(1)** a példányosított, adott CSV fájlnak megfelelő Java táblázat-objektum adott oszlopainak indexe tetszőleges sorrendben; **(2)** az oszlopok lefordítandó értéke; és **(3)** a fordítást jelölő oszlop indexe (nyilván ugyancsak 0-tól számozva).

Jól látható, hogy az áruház PDDL→CSV fordításakor 1 oszlopot fordítunk csak (az áruház PDDL objektum nevét). Tehát a lefordítandó oszlopok indexét magába foglaló tömb egyetlen elemű (csak a 0-ás index van benne), a lefordítandó szó a cselekvés 2-es számú argumentuma (azaz pl. `ZOLD`), és a fordítást az 1-es számú oszlopból vesszük ki (ez most pl. `green` lesz). Ehhez tehát érdemes egy pillantást vetni a `\jade\src\msclab01\planning_lab\csv\shopdict.csv` fájlra…

Ha megvan az áruház PDDL→CSV fordítása, akkor jöhet a tervbéli kosárba-cselekvés 1-es számú elemének, magának a terméknévnek a fordítása. Az is hasonlóképpen történik, csak itt most nem a 2-oszlopos áruház-szótárat használjuk, hanem a 3-oszlopos termék-szótárat (ahol tehát termék objektum névhez és CSV-s áruház névhez rendelünk egy-egy CSV-s terméknevet).

```
int[] inputidx2  = {0, 1};
String[] input2  = {action.elementAt(1), csvShop};
outputidx        = 2;
String csvProd   = proddict.translate(inputidx2, input2, outputidx);
```

Ezek után, ha ezt is sikerült lefordítani (nem üres a `translate` metódushívás értéke), akkor a `\jade\src\msclab01\planning_lab\csv\data.csv` fájlra építve elő is állíthatjuk az adott kosárba-cselekvésnek megfelelő URL-t:

```
int[] inputidx3  = {0, 2};
String[] input3  = {csvShop, csvProd};
outputidx        = 8;
url              = data.translate(inputidx3, input3, outputidx);
```

Itt tehát az áruház és a termék CSV-s nevéből/azonosítójából kiindulva keressük a `data.csv` azon sorát, ahol a 0. és a 2. oszlopok értéke rendre megyezik ezekkel, és aztán onnan kivesszük a 8. oszlop értékét, ami éppen az, ami nekünk kell, a PDDL cselekvés fizikai interpretációja, az URL.

Ezek után már csak végre kell hajtani ezt a fizikai interpretációt: küldeni kell egy HTTP REQUEST-et az adott URL-re.[42] Ezt a `PlanExecutorAgent.executeInterpretation` metódus végzi (aminek egyetlen bemenete az URL String).

[42] …ráadásul a HTTP RESPONSE-zal most nem is kell törődnünk, hiszen a PDDL modell kialakításánál feltételeztük, hogy tökéletesen meg tudjuk jósolni egy-egy cselekvés következményeit, és ezért nem érdekes számunkra

Az említett metódusok megfelelően ütemezett és paraméterezett hívását a `PlanExecutorAgent.ExecutePlan` JADE-viselkedés végzi – ezzel nem is kell foglalkoznunk. A hangsúly számunka – a saját PDDL modellünkhöz illeszkedő, LPG által kigenerált megoldási terv végrehajtatásakor – a `PlanExecutorAgent.interpretAction` metódus megfelelő átírása. A lényeg, hogy minden tervbéli cselekvésre készüljünk fel – azaz legyünk képesek mindegyiket megfelelő fizikai cselekvéssé konvertálni.

Az előbbi, 3 lépés hosszú absztrakt terv végrehajtása „fizikai” szinten a következő:

```
MODEL ACTION:  TO-CART 15TBSamsungHD154UISATAII32MBcachewinchesterECOgreen ZOLD
 REAL ACTION:  http://localhost/webshops/shop2?addtocart=7751

MODEL ACTION:  CHECK-OUT ZOLD
 REAL ACTION:  http://localhost/webshops/shop2/?checkout=1

MODEL ACTION:  TO-CART 1GB800MHzDDR2RAMGeilValueSPD5 KEK
 REAL ACTION:  http://localhost/webshops/shop1?addtocart=7501
```

III. Összefoglalás

Jelen anyagban elsősorban a Budapest Műszaki és Gazdaságtudományi Egyetem Méréstechnika és Információs Rendszerek tanszéke által szervezett „*Intelligens Rendszerek*” M.Sc. szakirány „*Kooperáció és gépi tanulás (VIMIM223)*” c. laboratóriumának „*Tervkészítés*” c. laborgyakorlatához kívántunk segédletet nyújtani.

Ennek során megismerkedtünk a tervkészítés elméleti és gyakorlati alapfogalmaival. A PDDL (Planning Domain Definition Language) nyelv lényegi elemeivel (a 2.2-es verzióig bezárólag), és filozófiájával, továbbá az LPG (Local search for Planning Graphs) tervkészítő alkalmazással (egész pontosan a jelenleg legfrissebb, „1.0-td” verzióval).

A megismert fogalmak és eszközök segítségével megoldottunk egy viszonylag összetett mintapéldát, aminek során mélyebb betekintést nyerhettünk a tervkészítés elméletébe és gyakorlatába, annak előnyeivel és buktatóival együtt. További elméleti és történeti útmutatót a függelék biztosít, ami az átfogó irodalomjegyzékkel együtt tanulmány szintjére emeli az anyag színvonalát.

Mindvégig kizárólag determinisztikus, részben rendezett tervkészítéssel foglalkozunk, habár a megoldott mintapélda nem támaszkodik túlságosan a tervkészítő algoritmus mikéntjére. A tervkészítő alkalmazást amolyan „fekete dobozként” használjuk megfelelően reprezentált tervkészítési problémák megoldására. Végső soron tehát a problémák reprezentációján, nem pedig megoldási módján van a hangsúly.

A tervkészítési mintapéldát követően részletesen kitértünk a labor szoftveres és hardveres infrastruktúrájának felépítésére és működésére. Ennek során a gyakorlathoz kapcsolódó feladatokból kifolyólag végső soron azzal foglalkoztunk, hogy egy PDDL modell alapján kigenerált terv „fizikai” interpretációjának és végrehajtásának mi a módja.

Az anyag elsajátításával viszonylag aktuális, és széleskörű ismereteket szerezhetünk az MI tervkészítés témaköréből. Némi gyakorlatra is szert tehetünk a példák implementációja során. Mindazonáltal az élvonal, a jelen kutatás, és a gyakorlat ennél is jelentősen előrébb tart. Példának okáért, érdemben nem is esik szó a nem klasszikus tervkészítésről, ahol a környezet nem feltétlen determinált (Younes és Littman, 2004), illetve a multi-ágens rendszerekben történő tervkészítésről (ahol az egyes ágensek nem az egyedüli tervkészítők, s így egymás lehetőségeit és céljait is célszerű mérlegelniük a hatékony működéshez). Ilyen értelemben tehát a jelen anyag csupán kiindulópontját képezheti az „MI tervkészítés” témakör teljesebb megismerésének és elsajátításának, esetleges további kutatásának.

IV. Függelék

A függelékben foglaltak ismerete nem előfeltétele a „Tervkészítés” c. laborgyakorlat teljesítésének, sokkal inkább elméleti adalék az érdeklődőbb hallgatók számára.

IV.1. Tervkészítés elmélete

Ebben a szakaszban a tervkészítés elméletével, és ennek korlátaival foglalkozunk. Áttekintjük az alapvetőbb terv-reprezentációkat és tervkészítési algoritmusokat, majd megvizsgáljuk a tervkészítési feladat komplexitását.

IV.1.1. Reprezentációk és Algoritmusok

A tervkészítési módszereket a problémamegoldó módszerektől általában a cselekvések, állapotok, célok és tervek reprezentációja különbözteti meg. Amíg a problémamegoldó módszerek *keresést* végeznek az állapotok vagy tervek terében, addig a tervkészítő módszerek esetében a terv megtalálására más-más, *specifikus módszerek* szolgálnak. A tervkészítés három fő alapgondolata:

- *Az állapotok, célok és cselekvések hozzáférhetővé tétele.* Ez azt jelenti, hogy a problémamegoldó módszerekkel ellentétben az ágens valamifajta formális leírást használ az állapotok, célok és cselekvések reprezentációjára. Az előbbi kettőt általában logikai mondatok formájában tárolja, míg a cselekvésekhez logikai elő és utófeltételeket rendel. Ez lehetővé teszi az állapotok és a célok közti közvetlen kapcsolatot, amennyiben az elő és utófeltételek környezeti állapotokra hivatkozó logikai állítások.
- *A tervkészítés függetlenítése a cselekvések végrehajtási sorrendjétől.* Ez azt jelenti, hogy a terv kialakítása során nem kell a cselekvések majdani sorrendjét követnünk a cselekvések tervbevételekor. A problémamegoldó módszerek stratégiája ellenben – a keresés következtében – mindig csak a majdani végrehajtás sorrendjében „fűzheti hozzá” a tervezett cselekvéseket a tervhez. A tervkészítés e szabadsága sokkal hatékonyabbá teheti a tervezést.
- *A világ független alkotóelemeinek felismerése.* Ez pedig azt jelenti, hogy a célok többnyire független részcélokra oszthatók, melyekre külön-külön részterv készíthető, s így a terv (megoldás) a résztervek (részmegoldások) összefűzésével állítható elő. A felbontást a reprezentáció formális jellege teszi lehetővé, nagyban megnövelve ezáltal a tervkészítés problémamegoldó módszerekhez viszonyított hatékonyságát.

A következőkben elvi sémákat mutatunk be különböző tervkészítő módszerek megvalósítására. Elsőként a részben rendezett tervkészítés (RRT), majd a módszer egy hatékony kiterjesztésének (RRT-DUNF) ismertetése kerül sorra, amit a hierarchikus dekompozíció (HD-RRT) bemutatása után a feltételes tervkészítés (FRRT) elvének leírása

zár. Mielőtt azonban még rátérnénk az algoritmusok tárgyalására, röviden ejtsünk szót a tervekkel kapcsolatos főbb alapfogalmakról.

A terv egyik legfontosabb alkotóeleme az operátor. Az *operátor* egy hármas adatstruktúraként fogható fel, melynek építőelemei a következők:

- *Cselekvés leírás:* az ágens-környezet számára szolgáló cselekvés-megnevezés.
- *Előfeltételek:* a cselekvés végrehajtásához szükséges környezetre vonatkozó állítások.
- *Következmények:* a cselekvés végrehajtása utáni környezetre vonatkozó állítások.

Legyen *o* egy operátor. Jelölje *o* előfeltételeit *elő(o),* következményeit pedig *köv(o).* Kezdetben csak olyan operátorokkal foglalkozunk, melyek előfeltételei ponált, míg következményei ponált és negált elsőrendű logikai kifejezések konjunkciójaként adottak, ahol a cselekvésleírás maga is egy logikai kifejezés (pozitív literál). Ekkor egy *o* operátor általános felírása a következő:

$$Op\big(\text{CSELEKVÉSLEÍRÁS: } cselekvés\_név(A_1, \ldots, A_n),$$
$$\text{ELŐFELTÉTEL: } elő_1\left(E_{1,1},\ldots,E_{1,k_1}\right) \wedge \ldots \wedge elő_p\left(E_{p,1},\ldots,E_{p,k_p}\right),$$
$$\text{KÖVETKEZMÉNY: } köv_1\left(U_{1,1},\ldots,U_{1,l_1}\right) \wedge \ldots \wedge köv_q\left(U_{q,1},\ldots,U_{q,l_q}\right)\big).$$

Látható, hogy *o*-ban *p* előfeltétel és *q* következmény szerepel, ahol a cselekvés egy *n*–argumentumú, az *i.* előfeltétel egy k_i–argumentumú, a *j.* következmény pedig egy l_j–argumentumú kifejezés. Az következmény-részben megállapodásunkhoz híven mind ponált, mind negált kifejezések szerepelhetnek.

Operátor-sémának nevezzük azt az operátort, amelyben nem minden kifejezés minden argumentuma *behelyettesített*, hanem létezik olyan kifejezés is, melynek valamely argumentuma *változó*. Az *o* operátort *alkalmazhatónak* nevezzük az ágens-környezet egy *s* állapotában, ha *o* változóinak létezik egy olyan lehetséges behelyettesítése, hogy a behelyettesített operátor előfeltételeire teljesül $elő(o) \subseteq s$. Az operátor végrehajtása után *köv(o)* összes ponált állítása igaz lesz *s* összes állításával egyetemben, kivéve azokat az *s*-beli (ponált) állításokat, amelyek *köv(o)*-ban negáltan szerepeltek. Az *s*-beli negált állításokat tehát mindaddig igaznak fogjuk tekinteni (implicite), amíg *s*-ben nem szerepel (explicite) az állítás ponált változata. (Ez egy igen „negatív”, ámde hatékony szemlélet).

Immár minden információ a rendelkezésünkre áll ahhoz, hogy definiálhassuk a terv fogalmát. A *terv* egy négyelemű adatstruktúra, melynek komponensei (Russell és Norvig, 2003):

- *A terv lépéseinek* halmaza, ahol a lépések a problémához kapcsolódó operátorok közül kerülnek kiválasztásra.
- *Sorrendi kényszerek* halmaza, ahol a kényszerek a terv lépéseinek egymásutániságát szabják meg.
- *Értékkényszerek* halmaza, ahol a kényszerek a terv lépéseiben (operátoraiban) szereplő változók értékeire tesznek megkötést.

- *Okozati kapcsolatok* halmaza, ahol a kapcsolatok a terv lépései közt fennálló ok-okozati kapcsolatokat írják le.

Legyen a terv lépéseinek halmaza *S*. Ekkor a terv valamely két $s_i,s_j \in S$ lépésre vonatkozó $\prec$ sorrendi reláció azt írja elő, hogy a terv majdani végrehajtásakor s_i meg kell, hogy előzze (nem feltétlen közvetlenül) s_j-t. Jelölje ezt a sorrendi kényszert: $s_i \prec s_j$.

Legyen V_t a *t* terv lépéseit leíró operátorokban szereplő változók halmaza. Az értékkényszerek ekkor valamely $V_t^i \in V_t$ változóra tehetnek megkötést oly módon, hogy vagy egy másik $V_t^j \in V_t$ változóval, vagy pedig egy konstanssal *egyesítik*.

Az okozati kapcsolatok a terv lépései közti ok-okozati összefüggéseket írják le. Ha $s_i,s_j \in S$ a *t* terv két olyan lépése, melyekre teljesül, hogy $\exists c \subseteq köv(s_i)$, melyre fennáll, hogy $c \subseteq elő(s_j)$, azaz az s_i lépés következményeinek valamely *c* részhalmaza szükséges előfeltétele s_j-nek, akkor azt mondjuk, hogy „s_i okozati kapcsolatban áll s_j -vel. Jelölje ezt az okozati kapcsolatot: $s_i \xrightarrow{c} s_j$.

A terveket tervkészítő módszerek segítségével állítjuk elő. A tervkészítés történhet egyrészt a lehetséges szituációk terében. Ekkor a tervet úgy állítjuk elő, hogy keresünk egy a kezdeti és a cél-állapot (szituáció) közt vezető cselekvés-sorozatot. Ha a keresés a kezdeti állapot felöl indul, akkor *progresszív tervkészítésről,* ha a cél-állapot felöl, akkor pedig *regresszív tervkészítésről* beszélünk. Másrészt a tervkészítés történhet a tervek terében is. Ekkor egy *részleges tervvel* indulunk, amit addig „bővítünk", amíg egy kész tervet nem kapunk belőle. A részleges tervek módosítása *finomító* és *módosító operátorok* segítségével történhet. A finomító operátorokkal a terv kényszereit, míg a módosító operátorokkal a terv bármely más aspektusát módosíthatjuk. *Részben rendezett* az a terv, amelyben bizonyos lépések sorrendje nem kötött egymáshoz képest. A részben rendezett tervek előállítására alkalmas tervkészítő módszereket *részben rendezett tervkészítőknek* nevezzük, ellenben a *teljesen rendezett tervkészítőkkel,* amelyek csupán olyan tervek előállítására képesek, melyekben a lépések sorrendje egyértelműen rögzített. Ezeket a terveket *teljesen rendezett* terveknek nevezzük. A részben rendezett tervek lépéseinek teljes sorrendezését a terv *sorbarendezésének* nevezzük. Hasonlóan az olyan tervet, amelyben csak behelyettesített változók fordulnak elő, *teljesen specifikált tervnek* nevezzük. Végül pedig, azt a tervet, amely garantálja egy *tervkészítő ágens* számára a cél elérését, az ágens-környezet által reprezentált probléma *megoldásának* nevezzük.

A megoldás lehet teljesen, vagy akár csak részben rendezett. Az utóbbi esetben lehetőségünk van bizonyos lépések párhuzamosítására, illetve a végrehajtási sorrend megválasztására. A részben rendezett tervek azonban nem mindig felelnek meg elvárásainknak. A következő két megkötésnek kell teljesülnie rájuk (Russell és Norvig, 2003):

(M1) $\forall s_j \in S$ *lépés* $\forall c \in elő(s_j)$ *előzményéhez* $\exists s_i \in S$ *úgy, hogy* $c \in köv(s_i)$ *és* $\neg\exists s_k$ *amelyre* $(\neg c) \in köv(s_k)$ *és* $s_i \prec s_k \prec s_j$ *teljesülhet.*

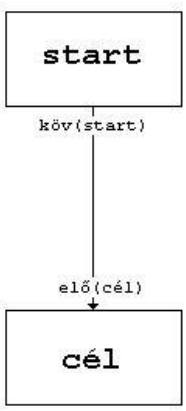
(M2) $\neg\exists s_{i_1}, s_{i_2}, \ldots, s_{i_N}$ *úgy, hogy* $s_{i_1} \prec s_{i_2} \prec \cdots \prec s_{i_{N-1}} \prec s_{i_N} \prec s_{i_1}$ *teljesül valamely* $N \in Z^+$ *esetén, továbbá a tervben szereplő semelyik V változóhoz sem tartozik egynél több különböző konstans értékkényszer.*

Az első feltétel (M1) a *teljességre* vonatkozik. E szerint minden lépés minden előfeltételéhez kell, hogy létezzen egy másik olyan lépés a tervben, amelynek az adott előfeltétel következménye. Továbbá nem létezhet olyan lépés, amely a végrehajtás során az előbbi két lépés közé „ékelődve" megszüntetheti az utóbbi megfelelő előfeltételét. A második feltétel (M2) a *konzisztenciára* vonatkozik. Eszerint se a terv sorrendi kényszereinek halmazában, se a változókra vonatkozó értékkényszerek (kötések) halmazában nem lehet ellentmondás.

Ha a fenti két feltétel teljesül egy tervre, akkor a terv megoldásnak tekinthető. A feltételek betartása pedig már a tervkészítő dolga.

RRT

Ebben a szakaszban a részben rendezett tervkészítés elvét vázoljuk röviden. A tervkészítés feladata, hogy a tervek terében egy teljes és konzisztens tervet, azaz egy megoldást találjon a problémára, amit kezdetben csupán csak egy *start* és egy *cél* lépés reprezentál. A *start* lépésnek nincsenek előfeltételei, csak következményei, amik a kiindulási állapotot állítják elő. A *cél* lépésnek pedig csak előfeltételei vannak, amik a kívánt cél-állapotot reprezentálják. Ezt nevezzük minimális részleges tervnek. Egy ilyen terv általános sémáját szemlélteti a IV-1. ábra.

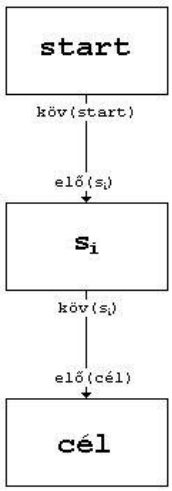


IV-1. ábra: Minimális részleges terv általános alakja

A tervkészítés során addig-addig finomítjuk a tervet, addig-addig adunk hozzá újabb és újabb lépéseket, amíg a *start* következményeiből a lépések segítségével a *cél* összes előfeltételét elő nem állítjuk, vagy a zsákutcába nem jutunk. Ekkor visszalépve másként próbáljuk meg kiegészíteni a tervet. Ha minden módon zsákutcába jutunk, akkor feladjuk.
Most pedig lássuk egy regresszív alapon működő részben rendezett tervkészítő vázlatos működését! Legyen adott – a IV-1. ábra szerint – egy probléma. A regresszív RRT algoritmus (röviden RRT) ezt tekinti kiindulási tervnek. Első lépésben megvizsgálja, hogy a terv megoldás-e. Ha igen, akkor készen van, ha nem, akkor kiválaszt egy olyan s_j lépést a tervből,

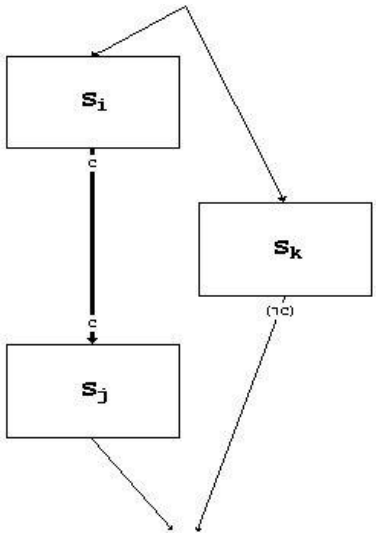
amelynek valamely $c \in elő(s_j)$ előfeltétele nem teljesül. Ehhez az előfeltételhez keres egy olyan s_i lépést (akár a terv lépései közül, akár az operátorsémák halmazából), amelynek következményei közt szerepel a c előfeltétel, azaz amelyre $c \in köv(s_i)$. Ha s_i a terv lépései közül való, akkor a terv okozati kapcsolatainak halmazához hozzáadja az $s_i \xrightarrow{c} s_j$ kapcsolatot, a sorrendi kényszerek halmazát pedig kiegészíti egy $s_i \prec s_j$ kényszerrel. Ha s_i még nem volt része a terv lépéseinek, akkor hozzáveszi, és a sorrendi kényszerek halmazát egy $start \prec s_i \prec cél$ kényszerrel egészíti ki. A kezdeti tervből kiindulva a fenti lépések végrehajtásával a IV-2. ábra által szemléltetett állapothoz jutunk.

Az ábrán látható tervben még csak sorrendi kényszerek láthatók (ezeket jelölik a vékony nyilak). A későbbiek során, újabb lépések behozatalát követően a tervben már okozati kapcsolatok (vastag nyilak) is megjelenhetek. Egy ilyen helyzetet mutat be a IV-3. ábra.



IV-2. ábra: A kezdeti terv egy lépéssel kiegészítve

Az ábrán egy olyan terv részlete látható, amelyben sérül az (M1) feltétel, hiszen s_k nem függ se s_i, se s_j lépésektől se sorrendileg, miközben a következményei közt szerepel egy olyan $(\neg c) \in köv(s_k)$ részállítás, amely *fenyegeti* az $s_i \xrightarrow{c} s_j$ okozati kapcsolatot. Ez azt jelenti, hogy s_k a végrehajtás során s_i és s_j lépések közé „ékelődve" megszüntetheti a $c \in elő(s_j)$ előfeltételt, amely szükséges feltétele az s_j lépés, s így az egész terv végrehajtásának. Az s_i és s_j közt fennálló okozati kapcsolatot ezúttal *védett kapcsolatnak* fogjuk nevezni. Az okozati kapcsolatok védelmét az RRT algoritmus az új lépések tervbevételét követően látja el. Teszi ezt úgy, hogy megvizsgálja az összes okozati kapcsolatot fenyegető lépés *előre*, vagy *hátramozdításának* lehetőségét.



IV-3. ábra: Terv-részlet egy fenyegetett okozati kapcsolattal

Előremozdításnak nevezzük azt, ha a terv sorrendi kényszereihez az $s_j \prec s_k$ kényszert, hátramozdításnak pedig azt, ha az $s_k \prec s_i$ kényszert adjuk. Az új sorrendi kényszer csak akkor válik véglegessé, ha hozzáadása nem sérti az (M2) feltételt, azaz nem okoz inkonzisztenciát a sorrendi kényszerek közt. *Lehetséges fenyegetésnek* számít továbbá s_k az $s_i \xrightarrow{c} s_j$ védett kapcsolatra nézve, ha s_k változóinak halmaza nem teljesen behelyettesített, s így létezik olyan (akár részleges) behelyettesítésük, hogy $(\neg c) \in köv(s_k)$ teljesülhet. Ennek a veszélynek a kezelésére több módszer is ismeretes, melyekre most külön nem térünk ki. Az RRT algoritmus működését szemléltető pszeudo-kód a következő:

```
function RRT(start, cél∈S, O) returns t∈T {
      t := Kezdeti_terv(start, cél)
      while ¬Megoldás(t) do {
            [sk, c] := Kielégítetlen_részcél(t)
            t := Terv_kiegészítés(t,O,sk,c)
            t := Fenyegetés_feloldás(t)
      }
      return t
}
```

Tehát a kezdeti tervből kiindulva, az operátorok *O* halmazának felhasználásával addig módosítjuk a tervet, amíg vagy egy megoldáshoz, vagy pedig zsákutcába nem jutunk. Először tehát keresünk egy még kielégítetlen részcélt, azaz a terv egy olyan s_k lépését, amelynek egy *c* előfeltételét még nem teljesítettük. Ilyen mindenképp kell, hogy legyen, hiszen *t*-ről megállapítottuk, hogy még nem megoldás. A következőkben tehát ennek a *c* előfeltételnek a kielégítése lesz a célunk, amihez a terv kiegészítéséhez kell folyamodnunk. Ha a kiegészítés minden módon meghiúsul, akkor zsákutcába jutottunk. Ha sikerül, akkor továbblépve megpróbáljuk feloldani a lehetséges (keletkezett) fenyegetéseket. Ha a fenyegetések feloldása nem oldható meg az (M1) és (M2) feltételek betartásával, akkor

visszalépve másként próbáljuk meg újra kiegészíteni a tervet. Ha nincs több lehetséges kiegészítés, vagy ha semelyik kiegészítés esetén sem tarthatók be az (M1) és (M2) feltételek, akkor zsákutcába jutottunk. Ha mégis sikerülne túljutnunk a fenyegetések feloldásán, akkor újra megvizsgáljuk, hogy a terv megoldás-e. Ha igen, akkor az a talált megoldással térünk vissza, ha nem, akkor újra kezdjük a kielégítetlen részcélok keresését.

Összefoglalva tehát azt mondhatjuk, hogy egy-állapotú problémák esetén az RRT megbízható, hatékony, helyes és teljes algoritmus. Sajnos azonban több-állapotú problémák esetén – az operátorok reprezentációjának „determinisztikus" jellege miatt – egy-egy tervbeli lépés többféle kimenetelének, vagy a lépések előfeltételtől függő különböző következményeinek a kezelése már nem megoldható. Az eshetőségi és a felderíthetőségi problémák általános kezelhetősége is kizárt, hiszen amellett, hogy esetükben is megjelenik a „nem-determinizmus", még a környezet és a cselekvések kimenetelének és reprezentációjának a bizonytalansága is közrejátszik.

RRT-DUNF

Ezeket a hiányosságokat részben pótolja az RRT algoritmus RRT-DUNF (R*észben* R*endezett* T*ervkészítés* D*iszjunkcióval,* U*niverzális kvantorokkal,* N*egálással és* F*eltételes következményekkel*) elnevezésű kiterjesztése, amely az operátor-leírás kifejezőbbé tételével próbálja meg rugalmasabbá tenni az eljárást. Ezek szerint az operátorok előfeltételei és következményei tetszőleges, konjunkciót, diszjunkciót és negációt tartalmazó univerzálisan kvantifikált logikai kifejezéseket lehetnek, ahol a következmények közt feltételes kifejezések is szerepelhetnek. A feltételes kifejezések *„Q* **when** *P"* alakú logikai állítások, melyekben *Q* akkor tekinthető az adott operátor egy következményének, ha a végrehajtás során a(z) – előfeltételekre (változóiban) hivatkozó – *P* állítás kiértékelése igaznak bizonyul.

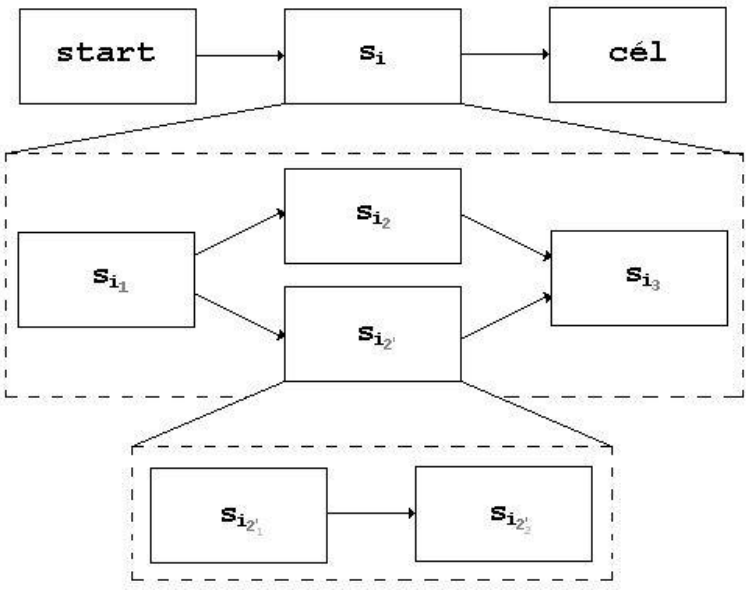
Az RRT-DUNF algoritmus a részletektől eltekintve gyakorlatilag megegyezik az RRT-vel. A különbség az algoritmus egyes lépéseinek megvalósításában, azaz a még kielégítetlen részcélok kiválasztásában, a terv kiegészítésének és a fenyegetések feloldásának módjában rejlik. A kielégítetlen részcélok kiválasztásakor ügyelnünk kell arra, hogy a kiválasztott részcélhoz tartozó tervbeli lépés következményei közt szerepel-e *„k* **when** *c"* alakú feltételes állítás, ahol *c* a lépés kielégítetlen előfeltétele. A részcélt akkor érdemes kiválasztanunk, ha *k* egy már meglévő okozati kapcsolat védett feltétele, hiszen ekkor *c* teljesülése (közvetve) szükséges feltétele *k* teljesülésének, s így a terv végrehajthatóságának. Ha a lépés következményei közt szerepel egy *„* $\neg k$ **when** *c"* alakú feltételes rész-következmény, akkor *c* teljesülése lehetséges fenyegetést jelent minden $s_i \xrightarrow{k} s_j$ okozati kapcsolatra nézve. A fenyegetés úgy oldható fel, ha biztosítjuk, hogy *c* nem teljesül. Ez úgy oldható meg, ha a terv kiegészítésekor csak olyan lépésések közül választunk, melyek következményei közt szerepel a $\neg c$ állítás. Ezt nevezik *konfrontációnak.* A *diszjunktív előfeltételek* sem okoznak gondot, csupán csak több lehetőséget kínálnak a kielégítetlen részcélok kiválasztására, hiszen a diszjunktív kifejezés bármely elemének teljesítése már elég a részcél teljesüléséhez. A *diszjunktív következmények* kezelése ellenben már sokkal bonyolultabb, hiszen azok az adott lépés több lehetséges kimenetelét írják le. Legtöbbször minden kimenetelre külön-külön tervet kell készítenünk, ami igencsak megnöveli a tervkészítés számítási igényét, hiszen a lehetséges tervek száma a tervek hosszával exponenciális arányban nő. További költséget

jelent az univerzális kvantifikáció bevezetése. Egy adott kifejezés változóira vonatkozó univerzális kvantifikáció megfelel a kifejezés összes behelyettesítése konjunkciójának. Ez – a „kifejtés” – mind az előfeltételekben, mind a következményekben rendkívül terebélyes logikai kifejezéseket eredményezhet. Továbbá megkötést kell tennünk a behelyettesítendő változók *típusára (típusolt argumentumok)* is, hiszen, ha a lehetséges behelyettesítések száma végtelen, akkor a konjunkció behelyettesített rész-állításainak az elemszáma sem lesz véges. Ehhez sajnos *statikus* környezetre van szükség, ahol a típusok véges száma már a kezdetek kezdetén adott s időben nem változik.

Összefoglalva elmondhatjuk, hogy bár az RRT-DUNF algoritmus a kifejezőbb leírásmódnak köszönhetően alkalmas lehet több-állapotú problémák megoldására, hatékonyságára és általánosságára nézve nincs garancia. Komplex, dinamikus környezetek esetén nemigen használható. Eshetőségi és felderíthetőségi problémák megoldása szóba sem jöhet általa. A leírásmód kellő megválasztása azonban nagyságrendi javulást eredményezhet az RRT-hez képest.

HD-RRT

A hatékonyság növelésének egy másik módja, ha nem a lépések (operátorok) leírását, hanem a tervben betöltött szerepüket értelmezzük át. A HD-RRT (H*ierarchikus* D*ekompozíciós* R*észben* R*endezett* T*ervkészítő*) alapötlete szerint hatékonyabb a tervet rész-tervekre bontva megoldani. Ehhez bevezeti az *absztrakt* és *elemi* operátorok fogalmát. Elemi operátoroknak nevezzük az operátorok O halmazának azon $E \subseteq O$ részhalmazát, melyben $\forall o \in E$ operátorra teljesül, hogy a terv végrehajtásakor közvetlenül végrehajtható cselekvést reprezentál. Az operátorok maradék $O \backslash E$ halmazának elemeit absztrakt (vagy összetett) operátornak nevezzük. A fogalmat az indokolja, hogy az absztrakt operátorok – mivel önmagukban még nem végrehajthatók – tekinthetők az őket tartalmazó terv rész-terveinek. Ahhoz tehát, hogy a terv végrehajtható legyen, az absztrakt operátorokat valahogy végrehajtható lépésekké (azaz résztervekké) kell alakítanunk. A *hierarchikus dekompozíció* során erre a célra *felbontási módszerek (sémák)* szolgálnak, melyek az absztrakt lépésekhez egy-egy résztervet rendelnek. Természetesen a felbontás nyomán előálló résztervekben is szerepelhetnek absztrakt lépések is, melyeket később hasonlóan helyettesíthetünk az őket megvalósító résztervekkel. A IV-4. ábra egy ilyen esetre mutat példát:



IV-4. ábra: Absztrakt lépések felbontása

Az ábrán egy – a tervben szereplő – összetett s_i lépés többszintű felbontása látható. A terv összetett lépéseit addig-addig bontjuk tovább a tervkészítés során, mígnem a kezdeti terv elemi lépések útján történő megvalósításához (azaz megoldáshoz) nem jutunk. A bemutatott gondolatmenetet a következő pszeudo-kód szemlélteti:

```
function HD-RRT(t, O, M) returns t∈T {
      while ¬Megoldás(t) do {
            [sk, c] := Kielégítetlen_részcél(t)
            t := Terv_kiegészítés(t,O,sk,c)
            si := Összetett_lépés_választás(t)
            t := Összetett_lépés_felbontás(t, M, si)
            t := Fenyegetés_feloldás(t)
      }
      return t
}
```

Több változtatás is történt az eddigi RRT algoritmushoz képest. Egyrészt megoldásnak immár csak az elemi lépésekből álló, (M1) és (M2) feltételeket kielégítő terveket tekintjük. Másrészt algoritmusunk most már nem csak egy *start-cél* lépésekből álló kezdeti tervet kaphat kiindulásként, hanem tetszőleges (akár kész) tervet is. Ennek előnye, hogy jobban „körvonalazható” a kívánt megoldás, több beleszólásunk van a végleges tervbe: részletesebben adhatjuk meg a kiindulást. Ezen felül két új művelettel is kiegészült az eljárás. A teljes algoritmus a következő:

Az eljárás három bemenetet kap: (1) a kezdeti t tervet, (2) az operátorok O halmazát és (3) a módszerek M halmazát, ahol a módszerek *felbontás(o,t')* alakú logikai állítások, melyek egy $o \in O$ absztrakt operátorhoz egy-egy olyan t' tervet rendelnek, amely az adott operátort valósítja meg.

Kezdetben megvizsgáljuk, hogy a kiindulási *t* terv megoldás-e. Ha még nem, akkor kell benne lennie egy még kielégítetlen *c* előfeltételt tartalmazó s_k lépésnek. Miután kielégítettük s_k előfeltételeit, megnézzük, hogy van-e még összetett s_i lépés a tervben. Ha van, akkor egy alkalmas (azaz s_i-re illeszkedő) felbontási módszer segítségével rész-lépésekre bontjuk, majd a *t* tervben ezekkel helyettesítjük. A terv meglévő értékkényszereihez hozzáadjuk a választott módszer értékkényszereit (változó-kötéseit). Minden *t*-beli $s_a \prec s_i$ sorrendi kényszert $s_a \prec s_m$ alakú sorrendi kényszerekkel helyettesítünk, ahol s_m a felbontás által adott részterv olyan lépése, melyhez nem létezik olyan s_j lépés a résztervben, hogy $s_m \prec s_j$ szerepelne a részterv sorrendi kényszerei közt. Magyarán s_a lépést a módszer „utolsó" lépései elé tesszük. Hasonlóan minden *t*-beli $s_i \prec s_z$ sorrendi kényszert $s_n \prec s_z$ alakú sorrendi kényszerekre cserélünk, ahol s_n a részterv olyan lépése, amely előtt más lépés már nem állhat, azaz amelyre a részterv sorrendi kényszerei közt nem létezik $s_{j'} \prec s_n$ alakú sorrendi kényszer. Ezen felül a *t* terv minden $s_a \xrightarrow{c} s_i$ okozati kapcsolatát cseréljük le olyan $s_a \xrightarrow{c} s_m$ alakú kapcsolatokra, ahol s_m a módszer által adott részterv olyan lépése, amely előtt nincs más olyan lépés a résztervben, melynek *c* előfeltétele. Hasonlóan minden $s_i \xrightarrow{c} s_z$ alakú *t*-beli okozati kapcsolatot helyettesítsünk $s_n \xrightarrow{c} s_z$ okozati kapcsolatok egy olyan halmazával, melyekben s_n a részterv olyan lépése, amely után már nincs olyan lépése a résztervnek, melynek *c* következménye. Ha az új értékkényszerek, sorrendi kényszerek vagy okozati kapcsolatok ellentmondást okoznak, akkor visszalépünk és másként próbáljuk meg felbontani s_i-t. Ha ez nem oldható meg, akkor zsákutcába jutott a tervkészítés. Ha sikerül, akkor megpróbáljuk feloldani a kapott tervben esetlegesen kialakult fenyegetéseket. Ha ez nem lehetséges, akkor visszalépünk és egy másik összetett lépést próbálunk meg ellentmondásmentesen felbontani. Ha nem találunk ilyet, akkor a tervkészítés végérvényesen zsákutcába jutott. Ha találunk, akkor újra megvizsgáljuk, hogy tervünk megoldás-e. Ha még mindig nem, akkor újra kezdjük a kielégítetlen részcélok kiválasztásánál, ellenben a kapott tervvel térünk vissza.

A hierarchikus dekompozíció elvét szinte minden ma ismert tervkészítő-rendszer alkalmazza kisebb-nagyobb változtatásokkal. Az alapelv egy analitikus, bizonyítottan helyes és teljes eljárást ecsetel. Alapesetben a HD-RRT képességei az RRT képességeivel azonosak, azaz az operátorok leírónyelvének kiterjesztése nélkül csak egyállapotú problémák megoldására alkalmas. Bármily meglepően is hangzik, általános esetben hatékonyság tekintetében sem jobb az RRT-nél. Ellenben – heurisztikák alkalmazásával – nagyságrendekkel hatékonyabbá tehető. Sajnos azonban ekkor már nem sokat mondhatunk a módszer képességeiről általánosságban. A heurisztikák közül talán a legismertebb a *lefelé* és a *felfelé megoldhatósági* tulajdonság. Az előbbi azt írja elő, hogy minden absztrakt *t* megoldáshoz kell, hogy létezzen olyan elemi megoldás, melynek *t* egy absztrakciója. Az utóbbi arról beszél, hogy elemi megoldásoknak nem létezik inkonzisztens absztrakciója. A felfelé megoldhatóság elégséges feltétele lehet az úgynevezett *egyedi elsődleges részcselekvés* betartása, amely szerint az absztrakt lépés felbontásában kell, hogy legyen legalább egy olyan lépés, melynek előfeltételei között szerepel az absztrakt lépés minden előfeltétele és következményei közt az összes következménye. Ezek szerint, ha egy terv egy adott felbontása inkonzisztens, akkor abból már nem lehet további felbontások útján megoldást produkálni, illetve, ha konzisztens, akkor biztosan létezik egy – további felbontás útján

előálló – megoldás. Sajnos mindkét feltétel teljesülésére létezik ellenpélda, így általános esetben sosem lehetünk biztosak abban, hogy egy inkonzisztens absztrakt terv elvetése egy konzisztens tervvel szemben jó-e, vagy sem, hacsak nem ismerjük kellő bizonyossággal cselekvéseink működését. Bonyolultabb (pl. eshetőségi, vagy felderíthetőségi) környezetben azonban erre nincs lehetőségünk, s így ekkor e heurisztikák alkalmazása sem jöhet szóba. Léteznek természetesen más egyéb heurisztikák is, amelyekkel szemben többnyire hasonló ellenvetéseink lehetnek. Komplex környezetek esetén az eljárás rugalmasságát az a tény is nehezítheti, hogy a tervkészítés módszereit (sémáit) készen kapja. Ezek ugyebár nem feltétlen felelnek meg a környezet elvárásainak, s így az általuk létrehozható terv sem feltétlen megfelelő.

FRRT

Ha tehát a környezet nem statikus és hozzáférhető, akkor az előzőleg bemutatott tervkészítési módszerek egytől-egyig csődöt mondanak. A probléma abban áll, hogy eshetőségi problémák esetén a cselekvések kimenetele, illetve a környező világ állapota nem jósolható meg. Eddig például megállapodás szerint úgy tekintettük, hogy azon logikai állítások, melyek nem szerepelnek egy-egy tervbeli lépés előfeltételei közt, mindenképp hamisak. Ez nem feltétlen igaz akkor, ha a környezet állapotára vonatkozó ismereteink hiányosak, s így az adott állításról nem áll módunkban nyilatkozni, avagy az állítás igazságtartalmára – cselekvéseink következményei alapján – nem tudunk biztosan következtetni. Elképzelhető tehát az is (jellemzően valós környezetek esetén), hogy nincs is olyan cselekvésünk, amely az adott – környezeti állapotra vonatkozó – állítást (mint előfeltételt) képes volna bizton előállítani.

Ennek megoldására született meg az FRRT (F*eltételes* R*észben* R*endezett* T*ervkészítés*) elve, mely szerint a terv végrehajtása során előforduló eshetőségekre előre felkészülhetünk, ha a tervkészítés során a tervbe feltételes elágazásokat, ún. *érzékelő cselekvéseket* iktatunk. Az érzékelő cselekvések döntik el, hogy a terv végrehajtásakor a lépések milyen *kontextusa*, azaz a környezet milyen állapota érvényes. Így, ha – a cselekvésekre és a környezetre vonatkozó megfelelő modellek birtokában – minden eshetőséget számításba tudunk venni, minden lehetséges kontextushoz külön-külön tervet készíthetünk, imígyen oldva meg az eshetőségi problémát. A tervkészítés ezen megközelítését szokás *feltételes*, avagy *eshetőségi tervkészítésnek* is nevezni.

A feltételes lépések IF-THEN-ELSE alakú adatstruktúrák, ahol a feltétel-rész egy érzékelő cselekvést valósít meg, a feltételes következmények pedig akár egész feltételes tervek is lehetnek. Ha a feltétel-rész által megvalósított érzékelő cselekvésnek előfeltételei is vannak, akkor azok előállítására külön (akár feltételes) tervet kell készítenünk. Ebben az esetben e terv képezi majd a feltételes lépés feltétel-részét.

Az FRRT algoritmus váza nagyjából megegyezik az RRT-évl. A főbb különbségek: (1) az értékkényszereken, sorrendi kényszereken és okozati kapcsolatokon túl immár a *feltételes kapcsolatok* is részei a tervnek. A feltételes kapcsolatok érzékelő lépéseket kapcsolnak össze a terv azon lépéseivel, melyek kontextusának egy része az adott érzékelésből származik. (2) az egyes tervbeli lépések kontextusa átöröklődik a későbbi lépésekre. (3) így a tervnek nem csak egy cél lépése lehet, hanem annyi, ahány különböző kontextus előállhat a tervben

szereplő érzékelések nyomán: minden eshetőségre külön tervet dolgozunk ki. (4) egy s_f lépés csak akkor jelent fenyegetést egy $s_i \xrightarrow{c} s_j$ védett kapcsolatra nézve, ha s_f és s_j kontextusai azonosak. Ezért a tervkészítés során a fenyegetések feloldását nem csak s_f előre, vagy hátramozdításával érhetjük el, hanem úgy is, hogy az s_j lépéssel feltételes kapcsolatban álló egyik s_{felt} megfigyelés ellenkező kimenetelét rendeljük hozzá s_f-hez feltételként, így különböztetve meg a két lépés kontextusát. Ezt nevezik *kondicionálásnak.* Az FRRT pszeudo-kódja a következő:

```
function FRRT(a0, C, O) returns t∈T {
      t := Kezdeti_terv(a0, C)
      while ¬Befejezés(t) do {
      C := Alternatív_célkörnyezet(t, C)
            [sk, c] := Kielégítetlen_részcél(t)
            t := Terv_kiegészítés(t,O,sk,c)
            t := Fenyegetés_feloldás(t)
      }
      return t}
```

A vázolt algoritmus a hasonlóságok ellenére lényegileg különbözik az RRT-től. Az eljárás bemenetére három paraméter érkezik. *O* az operátorok halmaza, *C* a kezdeti cél-konfigurációk (különböző kontextusú, ámde azonos előfeltételű cél-lépések) legalább egyelemű halmaza, míg a_0 a környezet érzékeléséből származó kezdeti állapot (logikai) leírása. A kezdeti állapot és a célok felhasználásával előállítjuk a kezdeti *t* tervet. Ezt befejezetnek tekintjük, ha a terv megoldás, azaz nincsenek benne kielégítetlen előfeltételek, teljes és konzisztens, továbbá a benne szereplő érzékelések minden kimeneteléhez létezik megfelelő kontextusú cél-lépés, azaz minden lehetséges cél-környezetét számításba veszünk. Ha *t* még nem befejezett, akkor előállítunk *t*-ben egy újabb cél-lépést, melynek kontextusa a meglévő cél-lépések kontextusai diszjunkciójának negáltja, azaz, amelynek kontextusa minden eddigi cél-kontextustól különböző. Ezt követően – az RRT-nél látottaknak megfelelően – keresünk egy még kielégítetlen előfeltételű tervbeli lépést, kielégítjük a terv kiegészítése által, majd végül feloldjuk a keletkezett fenyegetéseket. A fenyegetések feloldásánál immár – az előre és hátramozdításon túl – a kondicionálást is alkalmazzuk. A kondicionált lépések kontextusát tovább-terjesztjük a tervben egészen a nekik megfelelő célokig. Ha bárhol inkonzisztenciát tapasztalunk, akkor – az előző algoritmusok leírásánál látottaknak megfelelően – visszalépünk, és addig próbálkozunk, mígnem vagy kifutunk az összes lehetőségből, vagy egy befejezett tervhez nem jutunk. Ebben az esetben a kapott tervvel térünk vissza.

Összefoglalva, a feltételes tervkészítés elve alkalmas lehet eshetőségi problémák megoldására. A fentebb vázolt elvnek természetesen még több – hatékonyságot és rugalmasságot növelő – kiegészítése létezik. Ezek közül említésre szorul a *paraméterezett terv* fogalma, amelyben olyan megfigyelések is szerepelhetnek, melyek értéke egészen a terv végrehajtásáig ismeretlen marad. Ezen paramétereket *futásidejű változók* formájában követjük nyomon, amik tehát csak a terv végrehajtásakor nyernek behelyettesítést. A ciklusokat is tartalmazó feltételes cselekvések tekinthetők programoknak, melyek előállítására az *automatikus programozás* módszertana szolgál. Sajnos azonban – ha nem

zárjuk ki a rekurziók keletkezését – az algoritmus befejeződésére nem tudunk általános esetben garanciát adni. Látható, hogy – ha megengedjük az egyes érzékelő lépések többszöri felhasználását is a tervben – a feltételes cselekvések feltétel részében könnyen előállhat az a helyzet, hogy a feltétel-részt megvalósító tervben ugyanaz a feltétel önmagára hivatkozik (ördögi kör). A rekurziók kiküszöbölése, továbbá az összes eshetőség (cél-kontextus) figyelembevétele a tervkészítés során igencsak megnövelheti az eljárás erőforrásigényeit. A felderíthetőségi problémák – legalább közelítő – kezelése nem megoldott. A cselekvések, környezeti állapotok és érzékelések modelljeit sémák formájában továbbra is meg kell adnunk a tervkészítő számára.

IV.1.2. Elméleti nehézségek

Láthattuk, hogy a bizonyítottan helyes és teljes tervezés – azaz, amikor a visszaadott terv (ha létezik) mindenképp jó, „zsákutcába" jutva pedig biztosak lehetünk abban, hogy nincs megoldás – csak egyszerű környezetek, egy- vagy több-állapotú problémák esetén működött. Eshetőségi vagy felderíthetőségi problémák esetén legtöbbször már semmit se garantálhattunk. Ebben a szakaszban ennek megfelelően a következő kérdésekre keressük a választ: *Mi a tervkészítés bonyolultsága? Létezik-e terv, ami megoldja a problémát? Ha igen, végre lehet-e hajtani? Egyáltalán, mindig megválaszolhatók-e ezek a kérdések?*

Ahhoz, hogy megválaszoljuk e kérdéseket, először is be kell vezetnünk a komplexitással kapcsolatos főbb alapfogalmakat, majd pedig konkrétan definiálnunk kell, hogy mit is értünk pontosan „tervkészítés nehézsége" címszó alatt. Lényegében tehát két döntési-probléma kerül terítékre: egyrészt választ szeretnénk kapni arra, hogy mi a terv létezésének eldöntésére alkalmas algoritmus minimálisan szükséges lépésszám-igénye (TERV-LÉTEZÉS), másrészt arra, hogy mi annak az algoritmusnak a minimálisan szükséges lépésszáma, amely el tudja dönteni egy-egy tervről, hogy célba ér-e (TERV-VÉGREHAJTÁS).

Elsőként csak *környezet-specifikus* eseteket vizsgálunk. Ez azt jelenti, hogy egy-egy környezet-leírás viszonylatában, adott kiinduló-állapot mellett vizsgáljuk meg a két döntési probléma nehézségét. A szakasz végén röviden kitérünk majd a *környezet-független* eseteket feldolgozó elméleti eredményekre is, hivatkozás szintjén említve meg a számunkra fontosabbakat.

Mielőtt azonban rátérünk az eredmények összefoglalására, „nagy vonalakban" be kell vezetnünk a megértésükhöz szükséges főbb komplexitás-elméleti alapfogalmakat. Ehhez többnyire az irodalomban elterjedt jelölés-rendszerre (Papadimitriou, 1994) támaszkodunk.

Döntési problémának nevezzük annak az eldöntését, hogy egy adott *w* bemenet része-e egy $S = \{w \mid P(w)\}$ *P(w)* tulajdonságot kielégítő elemek halmazának. Egy *U* algoritmust akkor nevezünk polinom-idejűnek, ha a döntési problémára a bemenet hosszának polinomjával arányos számú lépésben képest választ adni, azaz – ha $|w|$ a bemenet hossza, és $t_U(w)$ az algoritmus minimálisan szükséges lépésszáma a *w* bemeneten, akkor – $t_U(w) \le p(|w|)$, ahol $p(|w|)$ a bemenet hosszának polinomja. Azon döntési problémákat, melyekre létezik

polinom-idejű algoritmus, a **P** probléma-osztály elemeinek tekintjük. Egy döntési probléma az **NP** osztályhoz (más jelöléssel a Σ_1P osztályhoz) tartozik, ha $w \in S$ eldöntése (azaz a *P(w)* tulajdonság teljesülésének vizsgálata) ekvivalens a $\exists u P(u,w)$ teljesülésének eldöntésével, ahol adott *u,w* esetén *P(u,w)* eldöntése polinom-időben megtehető. Az *u*-t a *w* „tanújának" nevezzük (Rónyai és társai, 1998). Hasonlóan, ha $w \in S$ eldöntése ekvivalens $\forall u P(u,w)$ teljesülésének eldöntésével, akkor a problémát **co-NP**-belinek, más szóval Π_1P-belinek nevezzük. Hasonlóan, egy döntési probléma Σ_kP-beli, ha eldöntése ekvivalens $\exists u_1 \forall u_2 \ldots P(u_1,u_2,\ldots,u_k,w)$ teljesülésének eldöntésével, és Π_kP-beli, ha eldöntése $\forall u_1 \exists u_2 \ldots P(u_1,u_2,\ldots,u_k,w)$ teljesülésének eldöntésével ekvivalens. Egy döntési probléma **PTÁR**-beli, ha $\exists k : k \le p(|w|)$ úgy, hogy a probléma Σ_kP vagy Π_kP-beli. Jelöljön *C* egy tetszőleges probléma-osztályt. Egy problémát akkor nevezünk *C*-nehéznek, ha minden *C*-beli probléma „redukálható rá", azaz létezik egy *Karp-redukció* (Rónyai és társai, 1998), amely bármely *C*-beli problémát képes polinom-időben visszavezetni az említett problémára. Egy probléma *C*-teljes, ha *C*-nehéz, és egyben *C*-beli is. Az említett problémák egy-egy probléma-osztály legnehezebbjei közé tartoznak. Meg kell még jegyeznünk azt, hogy, mivel valószínűleg $P \neq NP$, ezért – ahogy a tapasztalatok is mutatják – az NP-beli problémák megoldásához exponenciális, azaz közelítőleg 2^n-lépésszámú algoritmusok szükségesek, ahol $n = |w|$, a bemenet hossza. Hasonlóan, a Σ_2P, vagy Π_2P-beli problémák megoldásához már 2^{2^n} nagyságrendű lépésszámot igénylő algoritmusokra van szükség, nem is beszélve a PTÁR-beli problémákról… A P-hez, NP-hez és PTÁR-hoz hasonlóan definiálhatnánk L, NL és LTÁR, továbbá EXP, NEXP és EXPTÁR osztályokat is (Erol és társai, 1995), melyek már nem polinomiális, hanem logaritmikus, illetve exponenciális lépés-igény szempontjából osztályozzák a problémákat. Most pedig térjünk rá a tervkészítés komplexitás-elméleti eredményeinek vizsgálatára.

Elsőként tehát környezet-specifikus esetekben vizsgáljuk a tervkészítés nehézségét. Ezen belül a probléma-környezetek két fajtáját különböztetjük majd meg: *determinisztikus* és *véletlenszerű* környezeteket. Determinisztikus esetben a döntési probléma legyen a következő (TERV-LÉTEZÉS): Adott *D* környezet-leírás (amely tartalmazza a kezdeti állapotot és a cselekvések lehetséges kimeneteleinek leírását) és *f* cél ismeretében döntsük el, hogy létezik-e olyan *t* terv, amelyre $|t| \le p(|D|)$ teljesül (azaz ahol a tervhossz a környezet „komplexitásának" polinomjával becsülhető) és amely megvalósítja *f*-et. Az idevonatkozó eredményeket a IV–1. Táblázat foglalja össze.

IV-1. Táblázat: Tervezés bonyolultsága determinisztikus környezetben (Baral és társai, 1999)

Determinisztikus tervkészítési környezetek	
Döntési Probléma / **Terv típusa**	TERV-LÉTEZÉS
Teljes információ	NP-TELJES
Részleges információ, nincs érzékelés	Σ_2P-TELJES
Korlátos érzékelés	Σ_2P-TELJES
Korlátlan-érzékelés	PTÁR-TELJES
Részleges információ, teljes érzékelés	Π_2P-TELJES

A „teljes információ” azt jelenti, hogy mind a környezetleírás, mind pedig a cselekvések kimenetele teljesen ismert (*egyállapotú problémák*). A második vizsgált eset arra vonatkozik, amikor mind a környezetről, mind a cselekvések kimeneteléről csak hiányos információink vannak, továbbá a terveinkben nem szerepelnek érzékelő cselekvések (lásd. IV.1.1). A harmadik eset az előbbi eset véges számú érzékelés lehetőségével kiegészítve. A negyedik eset a harmadik esettel azonos eltekintve attól, hogy az érzékelések számára itt már nem szabunk korlátot. Az utolsó, ötödik esetben az előzőekhez hasonlóan, csak részleges információnk van a környezetről és a cselekvések lehetséges kimeneteleiről, ámde a környezet minden jellemzőjének érzékelése (pl. minden állapot-jellemzőt leíró állítás meglétének vizsgálata) lehetséges számunkra. Az eredmények sajnos sehol sem garantálnak polinom-idejű megoldást.

A véletlenszerű környezetek vizsgálati eredményeinek összefoglalásához definiálnunk kell a véletlenszerűséget kezelni képes algoritmusok, és a hozzájuk tartozó probléma-osztályok fogalmát. A PP (Probabilistic Polynomial) – az RP (Randomized Polynomial) probléma-osztályhoz „hasonlóan” – azon *S* nyelvek osztálya, melyekre $x \in S$ akkor, és csak akkor, ha létezik olyan polinom-tárkorlátos nem-determinisztikus *M* Turing-gép (azaz algoritmus), amely *x*-et többször fogadja el, mint ahányszor elutasítja. A PL (Probabilistic Logspace) probléma-osztályt hasonlóan definiálhatjuk, leszámítva, hogy ott egy logaritmikus-tárkorlátos Turing-gép meglétét kötjük ki. Adott *C* és *C'* tetszőleges probléma-osztályok esetén a $C^{C'}$ probléma-osztály azon nyelvek halmaza, amelyek *C-Turing redukálhatóak* a *C'* probléma-osztály nyelveinek halmazára. Az eddig felsorolt probléma-osztályok egymásba-ágyazottsága a következő (forrás: Littman és társai, 1998):

$$\mathrm{L} \subseteq \mathrm{NL} \subseteq \mathrm{PL} \subseteq \mathrm{P} \subseteq \frac{NP}{co-NP} \subseteq \mathrm{PP} \subseteq \frac{NP^{PP}}{co-NP^{PP}} \subseteq \mathrm{PTÁR} \subseteq \mathrm{EXP}$$

Az eddigiek ismeretében már definálhatjuk és értelmezhetjük a véletlenszerű környezetek esetén felmerülő döntési problémákat és hovatartozásuk. A TERV-VÉGREHAJTÁS problémája szerint (jelenleg) arra kell választ adnunk, hogy: Adott *D* környezet (amely tartalmazza a kezdeti állapotot, a lehetséges cselekvéseket, azok kimenetelét és a cél-állapotok explicite adott halmazát), *t* terv (melyre $|t| \le |D|$) és Θ küszöbszám mellett teljesül-e P(„*t* végrehajtása

cél-állapotba vezet a D környezetben")$>\Theta$ egyenlőtlenség? A másik probléma az előzőekben is látott TERV-LÉTEZÉS aktualizált változata, miszerint: Az előbbi módon adott D környezet, Θ küszöbszám és $z \leq |D|$ küszöbérték esetén teljesül-e P(„Létezik olyan z hosszú t terv, melynek végrehajtása cél-állapotba vezet a D környezetben")$>\Theta$ egyenlőtlenség? E döntések komplexitását mutatja az IV-2. Táblázat.

A táblázatban több, eddig definiálatlan fogalom is szerepel. A környezetet s*imának* nevezzük, ha a környezeti állapotok explicite fel vannak sorolva, míg *propozíciónálisnak*, ha az állapotokat propozíciónális logikai adat-struktúrákkal reprezentáljuk. Az utóbbi, bár sokkal tömörebb leírást biztosít, a TERV-LÉTEZÉS és a TERV-VÉGREHAJTÁS szempontjából sokkalta bonyolultabb (lásd. IV-2. Táblázat). Az említett két döntési probléma bonyolultságát – mindkét környezet-reprezentáció mellett – több különböző tervtípus esetén szemlélteti a IV-2. Táblázat. Ha a tervre vonatkozólag semmilyen megkötés sincs, vagy, ha a terv-hossz polinom-korlátos, akkor a TERV-VÉGREHAJTÁS problémájának bonyolultsága a környezet reprezentációjától függetlenül ismeretlen bonyolultságú. Ellenben – ugyanezen terv-típusok esetén – a TERV-LÉTEZÉS problémájára már akár polinom-rendű algoritmus is adható. Ez azért lehetséges, mert konkrétan kikötöttük a terv hosszát. A teljesen rendezett tervek vizsgálata nem jelent újdonságot, hiszen a IV.1. fejezetben már említettük őket.

IV-2. Táblázat: Tervezés bonyolultsága véletlenszerű környezetben (Littman és társai, 1998)

Véletlenszerű tervkészítési-környezetek				
Döntési Probléma / **Terv típusa**	Sima környezet-leírás		Propozíciónális környezet-leírás	
	TERV-VÉGREHAJTÁS	TERV-LÉTEZÉS	TERV-VÉGREHAJTÁS	TERV-LÉTEZÉS
Nincs megkötés	-	P-TELJES	-	EXP-TELJES
Polinom hosszúságú	-	P-TELJES	-	PTÁR-TELJES
Feltételes Ismétlődő	PL-TELJES	NP-TELJES	PTÁR-TELJES	PTÁR-TELJES
Feltételes Teljesen Rendezett	PL-TELJES	NP-TELJES	PP-TELJES	NP^{PP}-TELJES
Teljesen Rendezett	PL-TELJES	NP-TELJES	PP-TELJES	NP^{PP}-TELJES
Részben Rendezett (optimista)	NP-TELJES	NP-TELJES	NP^{PP}-TELJES	NP^{PP}-TELJES
Részben rendezett (átlagos)	NP-NEHÉZ	NP-TELJES	PP-TELJES	NP^{PP}-TELJES
Részben Rendezett (pesszimista)	CO-NP-TELJES	NP-TELJES	CO-NP^{PP}-TELJES	NP^{PP}-TELJES

Feltételes, teljesen rendezett tervek azok, amelyek teljesen rendezettek, de már – az FRRT-nél (lásd. IV.1.1) látott – feltételes lépések is szerepelhetnek bennük. Ezek kiterjesztései a feltételes ismétlődő tervek, melyekben egy-egy lépés többször is végrehajtódhat. A részben rendezett tervek vizsgálata három szemszögből történt: pesszimista, átlagos és optimista módon. Mivel a részben rendezett terveknek több sorba-rendezése is lehetséges, ezért gyakorlatilag teljesen rendezett tervek egy halmazának foghatók fel. Optimista esetben a részben rendezett tervet azon sorrendezése alapján ítéltük meg, amely a legnagyobb valószínűséggel eredményezett cél-állapotot. Pesszimista esetben a legrosszabb lehetséges

sorrendezés alapján ítéltük meg, míg átlagos esetben az összes lehetséges sorrendezés átlaga alapján. Az IV-2. Táblázatban összefoglalt eredmények szerint tehát – két speciális esettől eltekintve – véletlenszerű környezetekben legtöbbször nem adható determinisztikus (polinom-rendű) algoritmus se a TERV-LÉTEZÉS, se a TERV-VÉGREHAJTÁS problémájára függetlenül a terv típusától, vagy a környezet reprezentációjától.

Ha eltekintünk a környezet specifikációjától, akkor – környezet-független esetben – az IV-3. Táblázat foglalja össze a TERV-LÉTEZÉS problémájának eldönthetőségét:

Az táblázat szerint a funkció-szimbólumok bevezetése az operátor-leírásba – független az egyéb tényezőktől – általánosan **eldönthetetlenné** teszi a TERV-LÉTEZÉS problémáját, hacsak nem vezetünk be speciális megkötéseket (Erol és társai, 1995) a környezetre vonatkozólag. Ha a funkció-szimbólumok hiányában végtelen számú konstans megléte lehetséges, akkor a TERV-LÉTEZÉS problémája ugyancsak **eldönthetetlen** mindaddig, amíg nem zárjuk ki a több lehetséges kezdőállapot és a negált előfeltételek, vagy következmények lehetőségét. Az említett opciók majd mindegyikének kizárásával azonban a TERV-LÉTEZÉS problémája **eldönthetővé** tehető.

IV-3. Táblázat: Környezet-független TERV-LÉTEZÉS problémájának eldönthetősége (Erol és társai, 1995)

<table>
<tr><th>Lehetnek-e az operátorok leírásában funkció-szimbolumok?</th><th>Előfordulhat-e végtelen számú konstans?</th><th>Előfordulhat-e végtelen számú különböző kiindulási állapot?</th><th>Megengedettek-e negált előfeltételek és/vagy következmények?</th><th>A TERV-LÉTEZÉS problémája eldönthető-e?</th></tr>
<tr><td rowspan="2">igen</td><td>igen/nem</td><td>igen/nem</td><td>igen/nem</td><td>ELDÖNTHETETLEN</td></tr>
<tr><td>nem</td><td>nem</td><td>nem</td><td>ELDÖNTHETŐ</td></tr>
<tr><td rowspan="4">nem</td><td rowspan="3">igen</td><td>igen</td><td>igen/nem</td><td>ELDÖNTHETETLEN</td></tr>
<tr><td rowspan="2">nem</td><td>igen</td><td>ELDÖNTHETETLEN</td></tr>
<tr><td>nem</td><td>ELDÖNTHETŐ</td></tr>
<tr><td>nem</td><td>nem</td><td>igen/nem</td><td>ELDÖNTHETŐ</td></tr>
</table>

Összefoglalva az eddigieket, láthattuk, hogy összetettebb, valósabb környezetek és problémák esetén már a terv létezésének eldöntése sem oldható meg egykönnyen. Ha nem korlátozzuk a környezet, illetve a terv leírását, akkor analitikusan kezelhetetlen bonyolultságúvá, vagy akár eldönthetetlenné válhat a megfelelő terv meglétének (lásd. TERV-LÉTEZÉS) ellenőrzése. Nem-determinisztikus környezetekben már a terv megfelelőségének (lásd. TERV-VÉGREHAJTÁS) eldöntése sem oldható meg kivárható időn belül. A probléma bonyolultsága pedig – eddigi tudásunk alapján – nem megbecsülhető, hacsak nem teszünk komoly megkötéseket a terv struktúrájára nézve. Mindebből tehát az következik, hogy az „optimálitás” feltételeinek biztosítása – a TERV-LÉTEZÉS és a TERV-VÉGREHAJTÁS problémák bonyolultságának köszönhetően – általános esetben sajnos nem oldható meg analitikusan.

IV.2. Tervkészítés gyakorlata

Az előbbiekből kiderült, hogy a tervkészítés feladata eshetőségi és felderíthetőségi problémák esetén analitikusan kezelhetetlen bonyolultságú. A gyakorlatban alkalmazott tervkészítő rendszereknek valahogy mégis helyt kell(ett) állniuk, meg kell(ett) birkózniuk a valós környezet kihívásaival. A következőkben ilyen rendszerekre mutatunk példát, azaz jelentősebb gyakorlati tervkészítő módszereket és rajtuk alapuló tervkészítő alkalmazásokat sorolunk fel kronológiai sorrendben, a teljesség igénye nélkül.

IV.2.1. Történeti áttekintés

A tervkészítés gyökerei egészen a XX. század ’50-es éveiig nyúlnak vissza, mikoris Herbert Simon és Allen Newell kifejlesztették „LT (LOGIC THEORIST)” nevű rendszerüket (Newell és Simon, 1956). Ez volt az első rendszer, amely heurisztikákra építve feláldozta a „teljességet” a hatékonyság kedvéért. Propozícionális kalkulusban adott logikai tételek bizonyítására volt képes a tételekből történő *visszafelé-következtetés* által. Ehhez volt hasonló Gelernter „GTPM (GEOMETRY THEOREM-PROVING MACHINE)” rendszere (Gelernter, 1959), az első program, amely konjunktív rész-célokat is képes volt kezelni. A következtetéshez felhasználta a szimmetria tényét, és az előzőleg belátott tény-állításokat is. Az első jelentős próbálkozásnak mégis a „GPS (GENERAL PROBLEM SOLVER)” tekinthető (Newell és Simon, 1961). Bár „emberi intelligenciát” igénylő problémák megoldására tervezték, mégsem bizonyult annyira általánosnak, mint amilyennek szánták. Elsőrendű logikai bizonyításokat, logikai fejtörőket és egyéb „egyszerűen leírható” logikai feladványokat volt képes kezelni. Ez volt az első tervező-módszer, amely különválasztotta a problémára és a tárgy-tartományra vonatkozó tudást. Itt jelent meg először az *eszköz-cél* analízis módszere, amely során a rendszer az aktuálisan vizsgált „objektum” és a „cél” közti különbség alapján egy – táblázatból előkeresett – „cselekvést” kezdeményezett a különbség csökkentésére. Ha az adott cselekvés az adott objektumon nem volt értelmezhető, akkor – rekurzíve – próbálta meg a megfelelő „alakra” hozni. Hasonló – tétel-bizonyítási – elveken (újabban már a *rezolúciót* (Robinson, 1965) is latba vetve) működött Green „QA3” rendszere (Green, 1969), amely robot-irányítási feladatok megoldását tűzte ki célul. Ehhez először formalizálta a problémát, majd a tétel-bizonyító által megoldást keresett rá az állapot-térben. Az állapotokat a logikai állítások – „szituációkat” reprezentáló – változói szimbolizálták. A cél-feltétel egy egzisztenciálisan kvantifikált logikai állítás volt, melyet a tétel-bizonyító – egy megfelelő szituációt reprezentáló – változó-behelyettesítéssel próbált meg kielégíteni.

A tervkészítés „időszámítása” mégsem e rendszerekkel, hanem sokkal inkább Richard Fikes és Nils J. Nilsson „STRIPS (STANFORD RESEARCH INSTITUTE PROBLEM SOLVER)” nevezetű rendszerével (Fikes és Nilsson, 1971) kezdődött. „Vezérlési szerkezetét a GPS alapján alakították ki, az előfeltételek és következmények kezelésére pedig a QA3 egy változatát használták fel beépített modulként.” (Russell és Norvig, 2003). Leírásmódja lényegében a szituáció kalkulus és a Green által bevezetett formalizmus továbbfejlesztése, amit gyakorlatilag minden későbbi tervkészítő megörökölt. A dolgozatban vázolt RRT (lásd. IV.1.1) formalizmusához képest a megvalósítás fő különbségei, hogy a következmény rész

ponált és negált állításait különválasztva, egy *„add-list”* és egy *„delete-list”* formájában tárolja, továbbá, amíg az RRT részben rendezett, addig a STRIPS teljesen rendezett terveket állít elő. A STRIPS képes a különböző rész-célok lépései közti *átlapolódások* kezelésére is, így hát anno az elsők közt oldotta meg a „Sussman anomáliát” (lásd. II.2.1). Azonban eredeti változata a konjunktív célokból fakadó problémákból kifolyólag nem volt teljes. „A STRIPS vezette be a *háromszög-tábla* fogalmát, amely a teljesen annotált tervek egy hatékony ábrázolása” (Russell és Norvig, 2003). A STRIPS-nek volt továbbá egy – lényegében magyarázat-alapú – tanulási mechanizmusa is, amely tervek általánosításának tárolását tette lehetővé. Az így kapott terveket *„macrop”*-nak (mint makró-operátor) nevezték el, melyek lényege, hogy a tervben szereplő konstansokat lehetőség szerint változókkal helyettesítik. A STRIPS-et az SRI International kutatóintézet „Shakey” robot-projektjében alkalmazták először. A módszer bizonyítottan PTÁR-teljes (Canny, 1985; Chapman, 1987; Bylander, 1992).

Az említett „Sussman anomáliát” a „HACKER” nevezetű rendszer (Sussman, 1973) tesztelése kapcsán publikálták. A rendszer *képességek* megszerzését, épülését és fejlődését modellezte. Képességnek tekintjük a környezet valamely problémájának megoldására szolgáló eljárások halmazát. Ha új problémával szembesült, melyhez még nem volt megfelelő eljárás a birtokában, megtervezte azt. A visszalépéses technika helyett inkább *nyomkövető szakértőket* használt a rész-célok ütközéseinek feloldására. Így a STRIPS-szel egyetemben ez a rendszer is alkalmas volt a „Sussman anomália” orvoslására. Sajnos azonban nemhogy nem volt teljes, de még a megoldásul kapott tervek se voltak feltétlen „optimálisak”. Ellenben David Warren „WARPLAN” nevezetű rendszere (Warren, 1974) – előremozdítások és időbeni visszalépések segítségével (100 Prolog-sorban) – már képes volt a „Sussman anomáliára” „optimális” megoldást adni. A rendszer „WARPLAN-C” nevű változatában (Warren, 1976) jelent meg először a feltételes tervkészítés elve. Az „INTERPLAN” a (Tate, 1975) WARPLAN-hez hasonlóan képes volt a „Sussman anomália” megoldására, amihez úgynevezett *pillanat listákat* használt a védet kapcsolatok fenyegetéseinek feloldására, lehetővé téve a tervlépések közti tetszőleges átfedéseket.

E rendszerekkel nagyjából egy időben jelent meg az „ABSTRIPS (ABSTRACT STRIPS)” (Sacerdoti, 1974), amely korának első absztrakt, hierarchikus tervkészítője volt. Ez a módszer még nem a HD-RRT-nél (lásd .IV.1.1) látott hierarchikus dekompozíció elvét használta, hanem úgynevezett *közelítő hierarchiák* alapján kezelte az absztrakt operátorokat. Ennek lényege, hogy a tervkészítés – bár ugyanazon operátorok fordulnak elő a terv minden absztrakciós szintjén – az operátorok előfeltételeihez *fontossági szinteket* rendelve, a magasabb fontossági szintektől halad lefelé, azaz először csak a legfontosabb előfeltételeket veszi számításba, majd az így kialakított tervet tovább bővíti a kevésbé fontos előfeltételek hatásait is figyelembe véve egészen addig, mígnem már minden operátor minden előfeltétele szerepel a tervben. Bár az ABSTRIPS képes volt a hierarchikus tervkészítés megvalósítására, tervei még mindig teljesen rendezettek voltak.
Az első hierarchikus, részben rendezett tervkészítőnek a „NOAH (NETS OF ACTION HIERARCHIES)” tekinthető (Sacerdoti, 1975), amely – javítási műveletek végrehajtását támogató – szakértői rendszerként működött. Egyik módszere az *állapotkényszerítés* volt, amivel a környezetet – bizonytalanság esetén – egy ismert állapotba „kényszerítette”. A NOAH hierarchikus megoldásai még a közelítő hierarchiák elvén alapultak. A részben

rendezett tervek reprezentációjára *procedurális hálók* szolgáltak. Az úgynevezett *többes hatások táblázata (THT)* volt hivatott az egyes hálók csúcsai (terv-lépések) által „beiktatott és kitörölt" propozicionális logikai állítások nyilvántartására. A rész-célok közti kölcsönhatások az egyes állítások – egynél több csúcs általi – módosításának hatására jöttek létre, így a THT táblázatok figyelésével megoldható volt a fenyegetések (tervet veszélyeztető lépések) kiszűrése. A THT táblázatok felügyeletét *kritikusok* végezték, melyek a fenyegetéseket a terv átrendezésével oldották meg, továbbá kiszűrték a tervben többszörösen előforduló (redundáns) operátorokat és adott esetben változó-behelyettesítést is végeztek a tervkészítés elősegítésére. A NOAH a *legkisebb megkötés* elvét alkalmazta olyan tekintetben, hogy a tervet csak végszükség esetén rendezte sorba. A rendszer egy sikeres kiterjesztése volt a „NONLIN (NON-LINEAR)" (Tate, 1977), amely lehetővé tette heurisztikák alkalmazását, illetve *függőség-vezérelt* visszalépéssel egészítette ki a NOAH-t. A NONLIN volt az első olyan tervkészítő, „amely külön algoritmust használt az előfeltételek igazságértékének eldöntésére részleges tervekben" (Russell és Norvig, 2003). Elektromos turbinák generáljavításához használták.

Hasonló rendszerek jelentek meg más diszciplínák kutatási területein is. Molekuláris genetikai kísérletek tervezésére szolgált a „MOLGEN (MOLECULAR GENETICS)" (Stefik, 1981). A rendszernek két kulcspontja volt: egyrészt *kényszereket* vezetett be a részproblémák közti interakció explicit, deklaratív reprezentációjára, másrészt *több-szintű tervezést* tett lehetővé. Ez utóbbi három szintre volt bontható: a legmagasabb absztrakciós szinten, a *stratégia térben* az épp követendő tervkészítési stratégia került meghatározásra. Egy szinttel lejjebb, a *vázlat térben* a legalsó szint, a *tervezési tér* terveivel tervezhettünk, mintha csak egy meta-terv lépései volnának. A szintek közül csak a legalsó volt probléma-specifikus. A MOLGEN-hez hasonló kényszereket vezetett be a „SIPE (SYSTEM FOR INTERACTIVE PLANNING AND EXECUTION MONITORING)" is (Wilkins, 1988). Mint ahogyan a nevéből is sejthető, a rendszer lehetővé tette a terv végrehajtásának felügyeletét, s így az *újratervező rendszerek* előfutárának tekinthető. Ezen felül bevezette a tervezésbe az *erőforrás* fogalmát. Gyakorlatilag minden előfeltételt egy-egy bináris erőforrásnak tekintett, amely vagy szabad, vagy foglalt. Képes volt továbbá az erőforrásokat több különböző lépéshez is hozzárendelni. Egyéb előnyeinek és általánosságának köszönhetően felmerült komolyabb, valós tartománybeli alkalmazhatósága. Többek közt repülőgép-anyahajók leszállófedélzetének tervkészítési műveleteiben, többszintes épületek tervezésében és egy sörgyár gyártásszervezésében is segédkezhetett.

Ezzel nagyjából egy időben jelent meg a „TWEAK" (Chapman, 1987), amely a hierarchikus, részben rendezett tervkészítés első – bizonyítottan – helyes és teljes módszerei közé tartozik. Érdekességet jelent a tervek előállításának a módja, amely a *kényszer küldés* elvén alapszik. Ezek szerint a terv fokozatosan – újabb és újabb kényszerek beiktatásával – alakul ki. Újabb ugrást jelentett az „SNLP (SYSTEMATIC NON-LINEAR PLANNER)" (McAllester és Rosenblitt, 1991) megjelenése, amely bevezette a *szisztematikus* részben rendezett tervkészítés fogalmát. Szisztematikus tervkészítés alatt azt értjük, hogy a tervezés során egyetlen teljes vagy részben rendezett tervet se vizsgálunk meg egynél többször. A részben rendezett tervet itt definiálták először operátorok, okozati kapcsolatok és sorrendi kényszerek halmazának hármas adatstruktúrájaként. A rendszert gyakorlatilag a NONLIN egyszerűbb változatának *kiterjesztéssel* történő kiegészítésének tekinthetjük. A kiterjesztés – a rezolúció (Robinson,

1965) kiterjesztés lemmájának megfelelően – lényegében behelyettesített elsőrendű-logikai állításokon végzett rezolúciós bizonyításokhoz garantál olyan bizonyítást, mely ugyanezen állítások behelyettesítetlen változataira vonatkozik. Ezért a tervkészítés megengedi, hogy újabb és újabb operátorokkal bővítsük a tervet más tervbeli lépések előfeltételeinek kielégítése végett. Az előbbi megfontolásokat gyakorlatilag az RRT algoritmus foglalja össze.

Az RRT-ben használt leírónyelvet először az „O-PLAN (OPEN PLANNING)” nevű rendszer (Currie és Tate, 1991) egészítette ki az idő- és az erőforráskényszerek általános kezelésének módjával. Továbbá ez a módszer adott elsőként lehetőséget a HD-RRT-nél látott hierarchikus terv-reprezentációra és dekompozícióra. Heurisztikákat kínált a terv-keresés gyorsítására, miközben minden döntése okát regisztrálta. Jó tulajdonságai folytán szívesen alkalmazták (és alkalmazzák most is). Számos konkrét implementációja létezik a gyakorlatban: a gyártásszervezéstől a logisztikáig, az űrkutatástól a szoftverfejlesztésig. A kiterjesztések RRT-DUNF esetén (lásd. IV.1.1) látott letisztult egysége először az „UCPOP”-ban (Penberthy és Weld, 1992) jelenik meg. Az UCPOP egy hierarchikus, részben rendezett, többágenses tervezést támogató, feltételes következményeket is kezelni képes tervkészítő rendszer, az „ADL (ACTION DESCRIPTION LANGUAGE)” (Pednault, 1986) egyik legteljesebb megvalósítása. Máig ezt a – bizonyítottan helyes és teljes – módszert tekintik a részben rendezett tervkészítés kulcspontjának.

A ’90-es évektől napjainkig számos egyéb tervkészítő rendszer született. Számuk az utóbbi években – az igényekkel összhangban – rohamosan megnőtt, ezért a dolgozat, ha teljes felsorolásukra nem is, de a legfontosabbak közlésére próbálkozást tehet. A feltételes tervkészítés két alappillére az „UWL” (Etzioni és társai, 1992) és a „CNLP” (Peot és Smith, 1992), amelyek elvei az FRRT (lásd. IV.1.1) algoritmusban kerültek összefoglalásra. Valószínűségi következmények kezelésére is alkalmas feltételes tervkészítési módszer a „C-BURDIAN” (Draper és társai, 1994). Az „UMCP (UNIVERSAL METHOD COMPOSITION PLANNER)” (Erol és társai, 1994) a hierarchikus dekompozíció (akkoriban *HTN – „Hierarchical Task Network Planning”* néven emlegetett) elvének formális analízisére adott lehetőséget. Nagyjából ebben az időben rekedt meg a klasszikus módszerek továbbfejlesztése, és jelentek meg új eljárások. Ezek közül említésre méltó a „GRAPHPLAN” (Blum és Furst, 1995) és egy, a későbbiekben „SATPLAN”-nak elkeresztelt módszer (Kautz és Selman, 1996). Mindketten két fázisra bontották a tervkészítés folyamatát. Az előbbi egy *terv-gráfot*, míg az utóbbi *jól-formált propozíciónlális kalkulusbeli konjunktív normálformájú állítások* rendszerét hozta létre a probléma reprezentációjára, majd ebben „kereste meg” a megoldást. A két módszer előnyeit ötvözte a „BLACKBOX” (Kautz és Selman, 1998), amely a standard STRIPS-jelölésben megadott tervkészítési problémákat először terv-gráffá konvertálta, a terv-gráfot „CNF wff” formára hozta, majd a SATPLAN módszerével megoldotta. A BLACKBOX újabb verziói már PDDL (McDermott és társai, 1998) nyelven reprezentált problémákat is képesek voltak fogadni. E módszereket követte a hierarchikus dekompozíció – UMCP-nél látott – hagyományait felelevenítő „SHOP (SIMPLE HIERARCHICAL ORDERED PLANNER)” (Nau és társai, 1999), amely nagyságrendekkel hatékonyabbnak bizonyult a BLACKBOX-nál. A *sorrendezettség* azt jelenti, hogy a tervlépéseket a majdani végrehajtásukkal azonos sorrendben (mondhatni a részcélok szintjén rendezve a tervet) veszi figyelembe a (progresszív) tervkészítő.

Születtek azonban újszerűbb megközelítések is. A „MIPS (MODEL CHECKING INTEGRATED PLANNING SYSTEM)” (Edelkamp és Helmert, 2000) a modell-ellenőrzés módszerét vetette latba a tervkészítés során. Ez a rendszer használta először a *bináris döntési diagrammokat (BDD)* a környezet állapotainak tömör reprezentációjára. A „TLPLAN (TEMPORAL LOGIC PLANNER)” (Bacchus és Ady, 2001) a temporális-logikát hívta segítségül, lehetővé téve az „idő” fogalmának tisztán logikai alapon történő kezelhetőségét. A módszer a környezet – tetszőleges komplexitású – elsőrendű temporális-logikai reprezentációjának kezelésére képes – túl a tétel-bizonyításon – csupán a modell-ellenőrzés alapján. Az „FF (FAST-FORWARD PLANNING SYSTEM)” nevezetű rendszer (Hoffmann és Nebel, 2001) ezzel szemben – mondhatni klasszikus módon, az ABSTRIPS-hez hasonlóan – az operátorok – egyfajta heurisztikus – *relaxációjával* próbálkozik. A „VHPOP (VERSATILE HEURISTIC PARTIAL ORDER PLANNER)” (Younes és Simmons, 2002) a SHOP-pal ellenben már csak részleges sorrendezettséget kíván, hasonlóan a SHOP legújabb változatához, a „SHOP2”-höz (Nau és társai, 2003).

A legújabb tervkészítők, mint például az „LPG (LOCAL SEARCH FOR PLANNING GRAPHS)” (Gerevini és Serina, 2002), amely lényegében a GRAPHPLAN és a SATPLAN összetétele némi egyedi lokális keresési heurisztikával megtoldva, már a PDDL legújabb verzióinak kihívásaihoz mérten (Fox és Long, 2003; Edelkamp és Hoffmann, 2003; Gerevini és Long, 2005) numerikus, temporális, adott jósági mércéknek, hasznoknak megfelelő tervkészítésre is alkalmasak. Temérdek praktikus, és egyben elméleti kihívást is jelentő kiegészítéssel rendelkeznek (származtatott predikátumok, időzített kezdeti literálok, stb).

IV.2.2. Alkalmazások felsorolása

Az alábbi táblázatban az előbbi történeti áttekintésben (lásd. IV.2.1) szereplő tervkészítési módszerek, és a velük egybeforrt alkalmazások láthatók képességek szerint osztályozva, kronológiai sorrendben.

IV-4. Táblázat: Tervkészítési alkalmazások és módszerek osztályozása

TÍPUS / MÓDSZER	Hierarchikus		Nem hierarchikus		Feltételes tervkészítő	Kiterjesztett operátorok	Logika	
	Teljesen rendezett	Részben rendezett	Teljesen rendezett	Részben rendezett			0-rendű	1-rendű
LT							■	
GTPM							■	
GPS							■	
QA3							■	
STRIPS			■				■	
ABSTRIPS	■						■	
SIPE				■				
HACKER				■				
WARPLAN				■				
WARPLAN-C				■	■			
INTERPLAN				■				
TWEAK				■				
NOAH		■						
NONLIN		■						
MOLGEN		■						
SNLP		■						■
O-PLAN		■				■		■
UCPOP		■						■
UWL		■			■			■
CNLP		■			■			■
C-BURDIAN		■			■	■		■
UMCP		■						■
GRAPHPLAN		■						■
SATPLAN				■				■
BLACKBOX				■				■
SHOP	■							■
MIPS			■			■		■
TLPLAN			■			■		■
FF	■					■		■
VHPOP		■				■		■
SHOP2		■						■
LPG		■				■		■

IV.3. Egy bonyolultabb Sussman anomália és megoldása

Ebben a szakaszban a szemléletesség kedvéért a II.2. szakaszban bevezetett Sussman anomália PDDL-alapú (első körös) domain-leírásához mellékelünk egy valamivel bonyolultabb, konkrét PDDL-alapú probléma-leírást, illetve annak megoldását.

Bonyolultabb probléma-leírás

```
(define (problem sussman1_2)
(:domain sussman)
(:objects a b c d e f g h i j k)
(:init  (on a table)
        (on b table)
        (on d table)
        (on i table)
        (on j a)
        (on c b)
        (on h d)
        (on g i)
        (on e j)
        (on k c)
        (on f k))
(:goal  (and    (on a b)
                (on b c)
                (on c d)
                (on d e)
                (on e f)
                (on f g)
                (on g h)
                (on h i)
                (on i j)
                (on j k)
                (on k table))))
```

Bonyolultabb probléma megoldása

```
0:   (MOVE E J G) [1]
0:   (MOVE F K TABLE) [1]
1:   (MOVE K C TABLE) [1]
1:   (MOVE E G TABLE) [1]
2:   (MOVE G I E) [1]
2:   (MOVE J A K) [1]
2:   (MOVE C B TABLE) [1]
3:   (MOVE I TABLE J) [1]
3:   (MOVE F TABLE B) [1]
4:   (MOVE H D I) [1]
5:   (MOVE G E H) [1]
6:   (MOVE F B G) [1]
7:   (MOVE E TABLE F) [1]
8:   (MOVE D TABLE E) [1]
9:   (MOVE C TABLE D) [1]
10:   (MOVE B TABLE C) [1]
11:   (MOVE A TABLE B) [1]
```

Itt érdemes felfigyelni arra, hogy bizonyos időpillanatokban több cselekvés is végrehajtásra kerül párhuzamosan. Ennek oka, hogy a részben rendezett tervkészítő számára nem írtuk elő, hogy egyszerre csak és kizárólag egyetlen cselekvést szabad végrehajtani. Így tehát azon cselekvések, melyek sorrendje nem kötött egymáshoz képest, egyszerre kerül(het)nek végrehajtásra. …és ha valaki esetleg nem hiszi, hogy jó a terv, próbálja csak ki nyugodtan maga is!

IV.4. Sussman anomália másfajta megoldása

Ebben a szakaszban a Sussman anomália II.2.3-as szakaszban bemutatott PDDL-alapú reprezentációjától eltérő, „egyszerűbb” (második körös) domain- és probléma-leírását mellékeljük.

Másfajta domain-leírás

```
(define  (domain sussman2)
(:requirements :strips :equality)
(:constants      table)
(:predicates     (on ?x ?y)
                 (free ?x))

(:action move_to_cube
:parameters      (?cube ?from ?to)
:precondition    (and     (on ?cube ?from)
                          (free ?cube)
                          (not     (= ?cube table))
                          (not     (= ?cube ?from))
                          (not     (= ?cube ?to))
                          (not     (= ?from ?to))
                          (not     (= ?to table))
                          (free ?to))
:effect          (and     (on ?cube ?to)
                          (free ?from)
                          (not (on ?cube ?from))
                          (not (free ?to)))
)

(:action move_to_table
:parameters      (?cube ?from)
:precondition    (and     (on ?cube ?from)
                          (free ?cube)
                          (not     (= ?cube table))
                          (not     (= ?cube ?from))
                          (not     (= ?from table)))
:effect          (and     (on ?cube table)
                          (free ?from)
                          (not (on ?cube ?from)))
)
)
```

Másfajta probléma-leírás

```
(define (problem sussman2_1)
(:domain sussman2)
(:objects a b c)
(:init
         (on c a)
         (on a table)
         (on b table)
         (free b)
         (free c))
(:goal (and (on a b)
                 (on b c)
                 (on c table)))
)
```

A probléma megoldása pedig nyilván nem változik, hiszen annak ellenére, hogy más módon reprezentáltuk, végső soron még ugyanazt reprezentáltuk ekvivalens módon.[43]

[43] …bár ennek egzakt formális bizonyítására most itt nem vállalkoznék – elég, ha a józan ész látja, miről van szó (és nem mellesleg valóban ugyanazok a megoldások jönnek ki (amiket ugyanúgy interpretálhatunk)).

V. Irodalomjegyzék

Bacchus, F., Ady, M. (2001). *Planning with Resources and Concurrency: A Forward Chaining Approach*, In Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-2001), pp. 417-424

Baral, C., Kreinovich, V., Trejo, R. (1999). *Computational complexity of planning and approximate planning in presence of incompleteness.* In The International Joint Conferences on Artificial Intelligence (IJCAI).

Blum, A., Furst, M.L. (1995). *Fast planning through planning graph analysis*. Proc. IJCAI-95, Montreal, Canada.

Bylander, T. (1992). *Complexity results for serial decomposability*. In Proceedings of the Tenth National Conference of Artificial Intelligence (AAAI92), pp. 729-734. San Jose, California. AAAI Press.

Chapman, D. (1987). *Planning for conjunctive goals*. Artificial Intelligence, 32, pp. 333-377

Colmerauer, A. (1975) *Les grammaires de metamorphoses*. GIA, Univ. Marseille-Luminy, France, Tech. Rep.

Currie, K.W., Tate, A. (1991). *O-Plan: the Open Planning Architecture*. Artificial Intelligence, 52 (1), pp. 49-86.

Draper, D., Hanks, S., Weld, D.S. (1994). *Probabilistic Planning with Information Gathering and Contingent Execution*. In Kristian J. Hammond (ed.) Proceedings of the Second International Conference on Artificial Intelligence Planning Systems (AIPS-94), pp. 31-36. Morgan Kaufmann.

Edelkamp, S., Helmer, M. (2000). *On the implementation of MIPS*. Paper presented at the Fifth International Conference on Artificial Intelligence Planning and Scheduling Workshop on Model-Theoretic Approaches to Planning, Breckenridge, Colorado.

Edelkamp S., Hoffmann J. (2003). *PDDL2.2: The Language for the Classical Part of the 4th International planning Competition*, Technical Report No. 195, Institut für Informatik.

Erol, K., Nau, D.S., Handler, J. (1994). *UMCP: A Sound and Complete Planning Procedure for Hierarchical Task-Network Planning*, In AIPS-94, Chicago.

Erol, K., Nau, D.S., Subrahmanian, V.S. (1995). *Complexity, decidability and undecidability results for domain-independent planning*, Artificial Intelligence, Vol. 76, pp. 75-88.

Etzioni, O., Hanks, S., Weld, D., Draper, D., Lesh, N., Williamson, M. (1992). *An approach to planning with incomplete information*. In Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning.

Fikes, R.E., Nilsson, N.J. (1971). *STRIPS: a new approach to the application of theorem proving to problem solving*, Artificial Intelligence, 2 (3-4), pp. 189-208

Fox M., Long D. (2003). *PDDL2.1: An Extension to pddl for Expressing Temporal Planning Domains*, Journal of Artificial Intelligence Research 20: 61-124.
Gelernter, H. (1959). *Realization of geometry proving machine*. In Proceedings of an International Conference of Information Processing, pp. 273-282 Paris. UNESCO House.

Gerevini, A., Serina, I. (2002). *LPG: a Planner based on Local Search for Planning Graphs*, in Proceedings of the 6th International Conference on Artificial Intelligence Planning and Scheduling (AIPS-02), Toulouse, France.

Gerevini, A., Long D. (2005). *Plan constraints and preferences in PDDL3 - the language of the fifth international planning competition*, University of Brescia, Italy, Tech. Rep.

Green, C. (1969). *Theorem-proving by resolution as a basis for question-answering systems*. In Meltzer, B., Michie, D., and Swann, M. (eds.), Machine Intelligence 4, pp. 183-205. Edinburgh University Press, Edinburgh, Scotland.

Helmert, M. (2008). *Changes in PDDL 3.1*. http://ipc.informatik.uni-freiburg.de/PddlExtension

Hoffmann, J., Nebel, B. (2001). *The FF Planning System: Fast Plan Generation Through Heuristic Search*, In Journal of Artificial Intelligence Research, Vol. 14, pp. 253-302

Kautz, H., Selman, B. (1996). *Pushing the Envelope: Planning, Propositional Logic and Stochastic Search*, In Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96), pp. 1194-1201. MIT Press.

Kautz, H., Selman, B. (1998). *BLACKBOX: A New Approach to the Application of Theorem Proving to Problem Solving*. Workshop Planning as Combinatorial Search, AIPS-98, Pittsburgh, PA.

Littman, M.L., Goldsmith, J., Mundhenk, M. (1998). *The computational complexity of probabilistic planning*. Journal of Artificial Intelligence Research, volume 9, pp. 1-36

McAllester, D., Rosenblitt, D. (1991). *Systematic nonlinear planning*. In Proceedings of the Ninth National Conference on Artificial Intelligence (AAAI-91), Vol. 2, pp. 634-639. Anaheim, California. AAAI Press.

McCarthy, J. (1960). Recursive functions of symbolic expressions and their computation by machine. Communications of the ACM, Vol. 3, Issue 4, pp. 184-195.

McDermott, D., Ghallab, M., Howe, A., Knoblock, C., Ram, A., Veloso, M.; Weld, D., Wilkins, D. (1998). *PDDL---The Planning Domain Definition Language*. Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, New Haven, CT.

McDermott, D. (2000). The 1998 AI planning systems competition. AI Magazine 21, no. 2: pp. 35--55.

McGuinness, L. D., van Harmelen, F. (2004). *OWL: Web Ontology Language overview*. W3C Recommendation, http://www.w3.org/TR/owl-features/

Nau, D.S., Cao, Y., Lotem, A., Munoz-Avilla, H. (1999). *SHOP: Simple Hierarchical Ordered Planner*. In Proceedings of IJCAI-99.

Nau, D.S., Au, T.-C., Ilghami, O., Kuter, U., Murdock, W., Wu, D., Yaman, F. (2003). *SHOP2: An HTN Planning System*. Journal of Artificial Intelligence Research. *(being published)*

Newell, A., Simon, H.A. (1956). *The logic theory machine*. IRE Transactions on Information Theory, IT-2(3), pp. 61-79

Newell, A., Simon, H.A. (1961). *GPS, a program that simulates human thought*. In Billing H. (ed.), Lernende Automaten, pp. 109-124. R. Oldenbourg, Munich, Germany. Reprinted in Feigenbaum and Feldman (1963).

Papadimitriou, C.H. (1994). *Computational Complexity*. Addison-Wesley, Reading, MA.

Penberthy, J.S., Weld, D.S. (1992). *UCPOP: A sound, complete, partial order planner for ADL*. In Proceedings of KR-92, pp. 103-114

Peot, M., Smith, D. (1992). *Conditional nonlinear planning*. In Hendler J. (ed.), Proceedings of the First International Conference on AI Planning Systems, pp. 189-197. College Park, Maryland. Morgan Kaufmann.

Robinson, J.A. (1965). *A machine-oriented logic based on the resolution principle*. Journal of the Association for Computing Machinery, 12, 23-41.

Rónyai, L., Ivanyos, G., Szabó, R. (1998). *Algoritmusok*. Typotex, Budapest.

Russell, S.J., Norvig, P. (2003). *Artificial Intelligence: A Modern Approach*. Prentice Hall.
Sacerdoti, E.D. (1974). *Planning in a hierarchy of abstraction spaces*. Artificial Intelligence, 5 (2), 1, pp. 15-135

Sacerdoti, E.D. (1975). *The nonlinear nature of plans*. In Proceedings of the Fourth International Joint Conference on Artificial Intelligence (IJCAI75), pp. 206-214. Tbilisi, Georgia. IJCAII.

Stefik, M.J. (1981). *Planning and meta-planning.* Artificial Intelligence, 16, pp. 141-169.
Schwefel, H.P. (1995). *Evolution and Optimum Seeking*. Sixth-Generation Computer Technology Series. John Wiley & Sons, Inc., New York.

Sussman, G.J. (1973). *A computational model of skill acquisition*. Massachusetts Institute of Technology, Technical Report No. AI TR-297.

Tate, A. (1975). *Interacting goals and their use*. In Proceedings of the Fourth International Joint Conference on Artificial Intelligence (IJCAI-75), pp. 215-218, Tbilisi, Georgia. IJCAII.

Tate, A. (1977). *Generating project networks*. In Proceedings of the Fifth International Joint Conference on Artificial Intelligence (IJCAI77), pp. 888-893. Cambridge, Massachusetts. IJCAII.

Warren, D.H.D. (1974). *WARPLAN: a system for generating plans*. Department of computational logic memo 76, University of Edinburgh, Edinburgh, Scotland.

Warren, D.H.D. (1976). *Generating conditional plans and programs*. In Proceedings of the AISB Summer Conference, pp. 344-354

Wilkins, D.E. (1988). *Practical Planning: Extending the AI Planning Paradigm*. Morgan Kaufmann, San Mateo, California.

Younes, H.L.S., Simmons, R.G. (2002). *On the role of ground actions in refinement planning*. In Malik Ghallab, Joachim Hertzberg, and Paolo Traverso (eds.), Proceedings of the Sixth International Conference on Artificial Intelligence Planning and Scheduling Systems, pp. 54-61, Toulouse, France. AAAI Press.

Younes, H.L.S., Littman, M.L. (2004). *PPDDL1.0: An extension to PDDL for expressing planning domains with probabilistic effects*. Technical report, CMU-CS-04-167, Carnegie Mellon University.