Debreceni Egyetem
Informatikai Kar

# Tervkészítés

Témavezető:
Dr. Várterész Magda
egyetemi docens

Készítette:
Angyal Balázs
programtervező informatikus

Debrecen
2011.

# Tartalomjegyzék

**1. Tervezési modellek** **4**

1.1. Tervezés, mint keresési probléma . . . . . . . . . . 4
1.2. Tervezés, mint logikai következtetés . . . . . . . . . . 5
1.3. Klasszikus tervezés . . . . . . . . . . 6

**2. Reprezentációs módszerek** **6**

2.1. STRIPS . . . . . . . . . . 6
2.2. ADL . . . . . . . . . . 7
2.3. PDDL . . . . . . . . . . 8

**3. Tervező algoritmusok** **9**

**4. Tervkészítés állapottér-kereséssel** **10**

4.1. Progresszív keresés . . . . . . . . . . 11
4.2. Regresszív keresés . . . . . . . . . . 12
4.3. Értékelés . . . . . . . . . . 13

**5. Tervkészítés tervtérben való kereséssel** **13**

5.1. Teljesen rendezett tervkészítés . . . . . . . . . . 14
5.2. Részben rendezett tervkészítés . . . . . . . . . . 14
5.2.1. A POP algoritmus . . . . . . . . . . 16
5.3. Értékelés . . . . . . . . . . 20

**6. Tervkészítés nem determinisztikus problémakörben** **21**

6.1. Feltételes tervkészítés teljesen megfigyelhető világban . . . . . . . . . . 23
6.2. Feltételes tervkészítés részlegesen megfigyelhető világban . . . . . . . . . . 27
6.3. Végrehajtás monitorozás és újratervezés . . . . . . . . . . 30

**7. Ütemezési feladatok** **32**

7.1. Ütemezés erőforráskorlátokkal . . . . . . . . . . 34

**8. Alkalmazások** **36**

**9. Összefoglalás** **42**

**10. Irodalomjegyzék** **44**

# Köszönetnyilvánítás

Szeretnék köszönetet mondani Dr. Várterész Magda tanárnőnek, hogy ötleteivel és javaslataival segítette a szakdolgozatom elkészítését.

# 1. Tervezési modellek

Egy feladat elvégzésére irányuló lépések kialakítását tervkészítésnek nevezzük. A tervezés meghatározza mindennapjainkat: terv szükséges egy cég tevékenységeinek meghatározására, de akár a hétköznapi teendők megszervezését is ide sorolhatjuk.

A tervezési problémák közös tulajdonsága, hogy a célok a környező világhoz kapcsolódnak és ezek eléréséhez az adott világot tudnunk kell befolyásolni. Ehhez ismernünk kell magát a környezetet, azt, hogy az egyes cselekvések hogyan változtatják meg a világot, valamint alapvető következtetési ismeretekre is szükség van. Kétfajta megközelítés használható tervezéskor: az egyik általános problémák, a másik konkrét esetek megoldására felhasználható információkra és eljárásokra támaszkodik.

A szakdolgozat fő célja a tervkészítő módszerek és algoritmusok bemutatása, illetve értékelése abból a szempontból, hogy valós problémák esetén alkalmazhatóak-e és ha igen, milyen hatékonysággal.

## 1.1. Tervezés, mint keresési probléma

A tervezési problémákra a kutatások kezdetén keresési feladatként tekintettek. Erre épült a GPS (General Problem Solver) rendszer is, amelyet 1957-ben Herbert Simon, J. C. Shaw és Allen Newell készített. Univerzális problémamegoldó programnak szánták, de csak megfelelően formalizált egyszerű problémák megoldására volt képes. Valós feladatok esetén a kombinatorikus robbanás miatt a keresés kudarcba fullad. A mesterséges intelligenciában kombinatorikus robbanásnak hívjuk azt a jelenséget, amikor a lehetséges állapotok száma a keresés előrehaladtával hatványozottan nő.

A probléma szemléltetésére nézzünk meg példaként egy klasszikus feladványt: a Hanoi tornyait. Adott három rúd, amelyekre korongokat pakolhatunk. A kezdeti állapotban egy adott számú különböző méretű korong található az egyik rúdon úgy, hogy a legkisebb korong van legfelül, a második legkisebb korong alatta található és így tovább - a legnagyobb korong van legalul. Célunk, hogy az összes korongot ugyanebben a sorrendben átrakjuk egy másik rúdra, ám ezt úgy kell megtennünk, hogy közben bármelyik korong felett csak egy tőle kisebb méretű korong lehet.

A megoldás egyszerűnek tűnik: van egy kezdőállapot, vannak operátorok, amelyek a világ állapotát változtatják meg, van egy feltétel, ami ellenőrzi, hogy a célállapotot elértük-e; csak az odavezető utat kell megtalálnunk. Valójában ez gyakran komplikált feladat az alábbiak miatt:

– Elágazási tényező: bizonyos problémák (például a Hanoi tornyai 12 koronggal) esetén nagyon nagy lehet és ez könnyen megbéníthatja a keresőt.

– Cselekvések kölcsönhatása: egyes cselekvések kizárhatják egymást, ezért előfeltételeket kell megfogalmaznunk minden egyes operátorhoz.

– Cselekvések következménye: a legtöbb cselekvés nem juttat közelebb a célállapothoz. Tudni kell, hogy egy lépés valóban közelebb visz-e a megoldáshoz.

– Lokalitás: a cselekvések nagyobb hányada a világ kis részére van hatással. Ezt kihasználva több részcélra bonthatjuk fel az eredeti célt, ügyelve arra, hogy a megoldások könnyen összefésülhetőek legyenek.

## 1.2. Tervezés, mint logikai következtetés

A tervezési feladatok világát a szituációkalkulus (John McCarthy, 1969) hagyományos elsőrendű logikai formulák konjunkciójaként írja le, azzal a plusz információval kiegészítve, hogy az állítások egy adott állapothoz vannak rendelve. Példaként nézzük meg a Hanoi tornyai feladat alapállapotát:

$$\textit{(Rajta 1 A kezdo)} \wedge \textit{(Rajta 2 A kezdo)} \wedge \textit{(Rajta 3 A kezdo)}$$

Jelentése: *kezdo* állapotban az *1*-es, a *2*-es és a *3*-as korong az *A* rúdon van. A világ változásait leíró operátorok megadása egy *eredmeny(O, K)* függvény segítségével történik:

$$\forall \textit{korong, rud, allapot MOZGAT(korong, rud, allapot)} \Rightarrow \textit{(Rajta korong rud eredmeny(MOZGAT, allapot))}$$

Ez fejezi ki azt az állapotváltozást, amit a *K* állapotban az *O* operátor végrehajtása eredményez. Ahhoz, hogy a teljes állapotrendszer helyességét megőrizzük szükségesek lehetnek keret axiómák is, melyek azt írják le, hogy mi nem változik egy akció végrehajtása során. Az utolsó lépés a célállapot megfogalmazása, ezzel a feladat világát teljesen megadtuk:

$$\exists \textit{veg (Rajta 1 C veg)} \wedge \textit{(Rajta 2 C veg)} \wedge \textit{(Rajta 3 C veg)}$$

A probléma megoldása tételbizonyítással történik, a megadott logikai axiómákkal próbáljuk bizonyítani a célállapotban foglaltakat. A keresési problémához hasonlóan itt is több tényezőt kell figyelembe venni, egyébként a bizonyítás nem biztos, hogy sikeres lesz:

– A lokalitás elvét nem tudjuk kihasználni, ezért nagy számú keret axióma szükséges annak leírására, hogy mi nem változik a világban egy operátor végrehajtásakor.

– A bizonyításhoz szükséges idő exponenciálisan növekszik a megoldás hosszával.

– A következtetéshez nem használhatunk heurisztikát.

– A tervben felesleges lépések is előfordulhatnak.

## 1.3. Klasszikus tervezés

A tervezés ma is használt feltevései és alapfogalmai a klasszikus modellből származnak (Chapman, 1987 - Russel, 1994 - McDermott, 1995), ami egy teljesen kiszámítható, zárt világot feltételez diszkrét állapotokkal. Ez azt jelenti, hogy elég csak azokat az állításokat megadni az állapotleírásban, amelyek igazak - amit nem írunk le, azt hamisnak tekintjük.

Az állapotok leírása bináris állapotváltozókkal történik. Két kiemelt állapotunk van: egy kezdeti és egy célállapot. Egy probléma megoldását a kezdőállapotból a végállapotba juttató cselekvések sorozata fogja jelenteni. Változásokat csak cselekvések okozhatnak, ezeknek nincs hatása magára a tervezési folyamatra. A cselekvéseket operátorok segítségével írjuk le, beleértve a végrehajtáshoz szükséges előfeltételeket és a következményeket is.

Az eddigiek alapján kijelenthetjük, hogy a tervkészítés nem egyszerűsíthető le alapvető keresési problémára. Specifikusabb reprezentációra van szükség az elsőrendű logikánál, de korlátozni is kell a leíró nyelvet, hogy olyan tervező algoritmusokat lehessen használni, amelyek képesek hatékony következtetésekre.

# 2. Reprezentációs módszerek

A reprezentációk meghatározó tulajdonsága, hogy miként kezelik a hatásterjedés (ramification) és a minősítési (qualification) problémákat. Az első - keret problémaként is ismert - esetben a kérdés az, hogy a cselekvések lokális hatásait hogyan lehet globális következtetési módszerekkel figyelemmel kísérni. Az utóbbinál: hogyan lehet megadni egy cselekvés előfeltételeit? Ez azért érdekes, mert elvben végtelen számú előfeltételt fogalmazhatunk meg. Ahhoz, hogy kezelhető számú feltétellel dolgozzunk, ismernünk kell a cselekvések összes következményét.

## 2.1. STRIPS

Először a STRIPS[1]-ben (Richard Fikes - Nils Nilsson, 1971) jelent meg az a feltevés, hogy egy cselekvés végrehajtásakor csak a világ azon állapotai változnak meg, amit a cselekvés következményei megadnak. A reprezentáció részei:

– Állapotok leírása: A valós világot logika feltételekre osztjuk és az állapotokat pozitív literálok konjunkciójaként írjuk le. Ítéletkolgikai literál például a *Gazdag* ∧ *Boldog*, elsőrendű literál például az *(Ott* $\textit{Teherautó}_1$ *Szeged)*. Feltételezzük, hogy a világ zárt, azaz a nem felsorolt állapotok hamisak.

[1] STanford Research Institute Problem Solver

– Célok leírása: Az állapotokkal megegyező módon írjuk le őket. Egy $S$ ítéletlogikai állapot kielégíti a $C$ célt, ha $S$ tartalmazza az összes $C$-beli elemet. Példa: a *Gazdag* ∧ *Boldog* ∧ *Sikeres* kielégíti a *Boldog* ∧ *Sikeres* célt.

– Cselekvések leírása: Cselekvési sémákkal történik amelyek névből, előfeltételből és következményből állnak. Példa:

> *Cselekvés((Szállít t honnan hova),*
> *Előfeltétel: (Ott t honnan) ∧ (Teherautó t) ∧ (Raktár honnan) ∧ (Raktár hova),*
> *Következmény: ¬(Ott t honnan) ∧ (Ott t hova))*

A STRIPS nyelv megkötései (például, hogy a literáloknak függvénymenteseknek kell lenniük) azt a célt szolgálták, hogy a tervkészítő algoritmusok egyszerűbbek lehessenek, de idővel ez új problémákat eredményezett. Egyes valós problémákhoz nem elég kifejező a STRIPS, ezért született meg kiterjesztéseként az ADL[2] (Pednault, 1994) cselekvésleíró nyelv.

## 2.2. ADL

Az ADL nyelv újdonságai:

– Pozitív és negatív literálok is szerepelhetnek az állapotokban.

– A nem szereplő literálok meghatározatlanok - ezt nyílt világ feltételezésnek hívjuk.

– A célokban kvantoros változók is lehetnek.

– A célok konjunkciót és diszjunkciót is tartalmazhatnak.

– Használhatóak feltételes következmények ($T{:}S$ - $S$ érvényes, ha $T$ igaz).

– Egyenlőségek használata - például ($x{=}y$).

– A változóknak lehet típusa.

A példánkban szereplő Szállít cselekvés leírása ADL nyelvben az alábbira módosul:

> *Cselekvés((Szállít t: Teherautó honnan: Raktár hova: Raktár),*
> *Előfeltétel: (Ott t honnan) ∧ (honnan ≠ hova),*
> *Következmény: ¬(Ott t honnan) ∧ (Ott t hova))*

[2] Action Descriptor Language

## 2.3. PDDL

A PDDL[3] (Drew McDermott, 1998) a leíró nyelvek egységesítése miatt született meg és tartalmazza a STRIPS összes kiterjesztését. A tervezési problémákat két részre bontja: egy tárgyterületi részre, ahol a predikátumok és a cselekvések leírása található és egy probléma részegységre, ahol az objektumok, valamint a kezdő- és célállapot definíciók vannak. Ezek leírása a következőképpen történik:

*(define (domain <domén név>)*
*<predikátumok leírása>*
*<cselekvések leírása>)*

*(define (problem <név>)*
*( :domain <domén név>)*
*<objektumok leírása>*
*<kezdőállapot leírása>*
*<célállapot leírása>)*

A cselekvések következményeinek definiálásakor alkalmazhatunk univerzális kvantálást és feltételes hatásokat is megadhatunk:

*(forall ( ?v1 ... ?vn)*
*<kövezkezmény>)*

*(when <feltétel>*
*<következmény>)*

Konkrét példaként nézzük meg, hogy hogyan adhatjuk meg egy Shakey világa problémának a leírását. A feladatban adott három szoba, amelyeket folyosó köt össze, három doboz és egy Shakey nevű robot. Shakey a következő cselekvéseket tudja végrehajtani: a szobák között tud mozogni, a villanyt fel tudja kapcsolni, a dobozokat el tudja tolni és fel tud rájuk mászni. A feladat kezdőállapotában mindegyik doboz az egyes számú szobában van és Shakey a hármas szobában tartózkodik. Célunk, hogy az egyes jelzésű doboz a kettes szobába kerüljön és ugyanitt fel legyen kapcsolva a villany. A probléma specifikációjának elkészítéséhez az alábbi formálizációkat kell elvégezni (az így előálló teljes feladatleírás a 8. fejezetben található):

– Objektumok: a három szoba, a három doboz és Shakey (*szoba1*, *szoba2*, *szoba3*, *doboz1*, *doboz2*, *doboz3*, *shakey*).

[3] Planning Domain Definition Language

– Predikátumok: *x* egy hely? *x* egy robot? *x* az *y* helyen van? [...] (*(SZOBA ?x)*, *(ROBOT ?x)*, *(ott ?x ?y) [...]* ). A *(ROBOT x)* predikátum igaz, ha *x* egy robot.

– Kezdőállapot: mindhárom doboz az egyes szobában van. Shakey a kettes szobában van. [...] (*(ott doboz1 szoba1)*, *(ott doboz2 szoba1)*, *(ott doboz3 szoba1)*, *(ott shakey szoba2) [...]* ). Az itt felsorolt állítások mindegyike igaz, amiket nem írtunk le, azokat hamisnak vesszük.

– Célállapot: az egyes számú doboz a kettes szobában van. [...] (*(ott doboz1 szoba2) [...]* ). Csak a felsorolt állításoknak kell teljesülniük, a többi körülménnyel nem foglalkozunk.

– Cselekvések: Shakey fel tud mászni egy dobozra. [...] A tevékenység leírása a *mi*, *honnan* és a *hova* paraméterhármassal történik. A végrehajtáshoz a következő előfeltételeknek kell teljesülniük: *(ott ?mi ?honnan), (ott ?hova ?honnan), (DOBOZ ?hova), (ROBOT ?mi)*. A cselekvés hatásaként pedig Shakey a dobozon lesz a padló helyett, tehát a *(rajta ?mi ?hova)* igaz lesz, a *(padlon ?mi)* pedig hamis. [...]

# 3. Tervező algoritmusok

A tervező algoritmusok három bemenettel dolgoznak:

– A világ leírása.

– A cselekvő személy céljának leírása abból a szempontból, hogy milyen viselkedést várunk el tőle.

– Elvégezhető cselekvések leírása.

Ezek mindegyike valamilyen formális nyelv segítségével van megadva. A tervezőprogram kimenete egy olyan cselekvéssorozat lesz, amelyet bármely világban végrehajtva elérjük a megadott célt feltéve, hogy a világ kezdőállapota megegyezik a feladatban megfogalmazott kezdőállapottal. A tervezési probléma egy ilyen fajta reprezentációja nagyon absztrakt.

Általánosságban elmondható, hogy léteznek sokkal kifejezőbb nyelvek a tevékenységek, állapotok és világok definiálására. Ám minél kifejezőbb nyelvet használunk, annál nehezebb a tervező algoritmust megírni és az eljárás végrehajtási sebessége is csökken. Ezért ezeket az egyszerűsítő feltevéseket alkalmazzuk:

– Egy cselekvés végrehajtását nem lehet megszakítani, így figyelmen kívül hagyhatjuk a világ azon állapotait, amelyekben folyamatban lenne egy cselekvés. Egy tevékenység

elvégzését állapotok közötti elemi transzformációként foghatjuk fel, két cselekvést egy időben nem hajthatunk végre.

– Minden akció hatása és maga a világ is determinisztikus.

– Az cselekvő teljes ismerettel rendelkezik a kiinduló állapotról és a világ egészéről.

– Csak a cselekvő befolyásolhatja a világot, ami alapvetően statikus. Ez azt követeli meg, hogy a tervező kezdeti bemenete a kezdőállapotot.

Ezek a feltételezések nem realisztikusak, de nagy mértékben egyszerűsítik a tervezési problémát, hogy egyszerű eljárásokkal tudjunk dolgozni. Először ilyen algoritmusokat nézünk meg, de később jóval bonyolultabbakat is áttekintünk, amelyek már nem használják ezeket a megszorításokat.

# 4. Tervkészítés állapottér-kereséssel

Tervezőt legegyszerűbben úgy tudunk készíteni, hogy a feladatot leképezzük a világ állapotai között történő keresésre. Az állapotteret gráfként foghatjuk fel, ahol minden csomópont egy állapotot képvisel. Élek kötik össze azokat az állapotokat, amelyek egy cselekvés végrehajtásával elérhetőek. Általánosságban az élek irányítottak, de a legtöbb probléma esetében minden tevékenység „visszavonható”, ezért célszerű kétirányú éleket használni a gráfokban. A feladat megoldása tulajdonképpen egy út az állapottér-gráfban.

A keresési problémaként való felfogás előnye, hogy egyszerűen alkalmazható minden általános „nyers-erő” módszer és heurisztika. Használhatunk szélességi vagy mélységi keresőt, de ha szükséges, akkor bonyolultabb memória korlátos algoritmusokkal is dolgozhatunk (Russel - Korf, 1992).

Most vizsgáljuk meg az állapottér struktúráját. Ehhez a legkézenfekvőbb egy olyan tervezőt készíteni, amely nem determinisztikus algoritmussal operál. Ezt a legegyszerűbben úgy tehetjük meg, hogy a tervező algoritmusba egy nem determinisztikusan viselkedő 'választ' metódust implementálunk, ami véletlenszerűen fog választani a megadott operátorok halmazából.

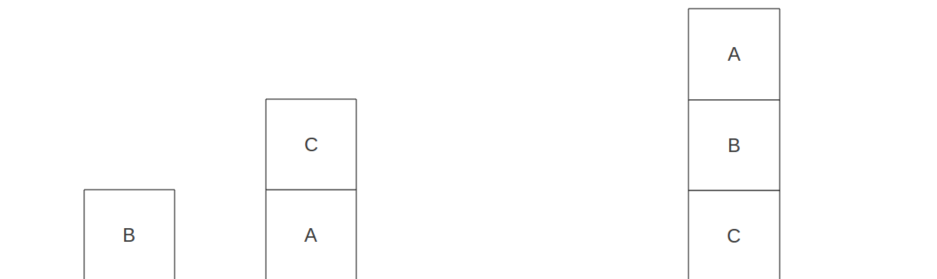
Előnye, hogy bármilyen agresszív vagy becslésen alapuló stratégiát szimulálhatunk vele, hiszen azzal, hogy elhatároltuk a keresési stratégiát ettől az alapvető algoritmustól, két dolgot érünk el: az eljárás egyszerűbb, könnyebben érthetőbb lesz és a sebesség érdekében tetszés szerint változtathatunk a keresési stratégián.

## 4.1. Progresszív keresés

Konkrét példaként nézzünk meg egy olyan algoritmust, ami előrefelé keres a megadott kezdőállapottól addig, amíg olyan állapotot nem talál, hogy az kielégíti a célfeltételeket.

ProgKereses(*allapot,celok,operatorok,utvonal*):
**Ha** az *allapot* tartalmazza az összes *celok*ban található elemet
**akkor return** *utvonal*
**különben** válasszunk az operátorok közül egy olyan cselekvést = *CS*
aminek az előfeltételei teljesülnek az allapotban
  **Ha** ilyen nincs **akkor return** nincs megoldás
  **különben** legyen *S* az az állapot amelyet az operátor alkalmazásával kaptunk
  **és return** ProgKereses(*S,celok,operatorok,hozzaad(utvonal,CS)*)

Egy nem determinisztikus algoritmus viselkedését úgy képzelhetjük el, mintha magunk választanánk ki a cselekvéseket a ’választ’ függvény meghívásakor. Az algoritmus szemléltetéséhez vegyük példaként a Sussman Anomália feladatot a kockavilágban, ahol van egy asztallapunk és 3 darab kockánk *A*, *B* és *C*. A kezdeti állapotban az *A* és a *B* jelű kocka az asztalon van, a *C* pedig az *A*-n. Célunk, hogy mindhárom kocka egymáson legyen fentről lefelé a következő sorrendben: *A*, *B*, *C*.




1. ábra: A Sussman Anomália kezdő- és célállapota

A függvény első meghívásakor az *allapot* a kezdőállapotot fogja tartalmazni, a *celok* a *(rajta A B)* $\wedge$ *(rajta B C)* konjunkciót, az *utvonal* pedig a null értéket. Mivel a kezdőállapot nem elégíti ki a célállapotot, ezért meghívásra kerül a ’választ’ metódus. Ebben a helyzetben a legkézenfekvőbb a ’tegyük C-t az A-ról az Asztalra’ cselekvés, ezért feltételezzük, hogy ezt választja a számítógép. Ha mindig a legjobb választást hajtanánk végre, akkor három lépésben eljutnánk a megoldáshoz és a program minden esetben kiléphetne, ha zsákutcába kerül, mert ez azt jelentené, hogy nem létezik az adott feladatnak megoldása.

Nyilvánvalóan, ha számítógéppel dolgozunk, muszáj keresést alkalmaznunk. Az algoritmus implementálásakor célszerű a 'választ' eljárást szélességi kereséssel megoldani. Ez minden esetben garantálja, hogy megtaláljuk a megoldást, ha létezik.

## 4.2. Regresszív keresés

Hátraláncoló kereséskor a céltól haladunk a kezdeti állapot felé, tehát olyan cselekvéseket kell kiválasztanunk, amelyek hatása megegyezik a célok halmazának egyik elemével.

RegKereses(*alapallapot,jelenlegicelok,operatorok,utvonal*):
**Ha** az *alapallapot* tartalmazza az összes *jelenlegicelok*ban található elemet
**akkor return** *utvonal*
**különben** válasszunk egy olyan cselekvést = *CS* amely hatása megtalálható a célok között
*C* = *jelenlegicelok* regresszáltja *CS*-vel
  **Ha** nem sikerült választani **vagy** *C* nem meghatározott **vagy** $C \supset$ *jelenlegicelok*
  **akkor return** nincs megoldás
  **különben return** RegKereses(*alapallapot,C,hozzaad(utvonal,CS)*)

A regresszálás eredménye egy olyan logikai mondat lesz, amelybe bekerülnek a kiválasztott cselekvés előfeltételei és törlésre kerülnek a cselekvéshez tartozó célok. Formálisan: *CS* előfeltételei $\cup$ (*jelenlegicelok* - *CS*-hez tartozó célok).

Kezdetben a *jelenlegicelok* tartalma: *(rajta A B)* $\wedge$ *(rajta B C)*. A 'választ' függvény meghívásakor olyan cselekvést kell választani, amelynek a hatása *(rajta A B)*. Ez a feltétel a 'tegyük az asztalról A-t B-re' operátor esetén teljesül, így ezt választjuk. Az operátor előfeltétele *(rajta A Asztal)*, következménye *(rajta A B)* $\wedge$ *-(rajta A Asztal)*, tehát a regresszió eredménye a következő lesz: *(rajta A Asztal)* $\wedge$ *(rajta B C)*.

Ha nem tudunk kiválasztani egyetlen cselekvést sem, akkor zsákutcába futottunk. Ennek az ellenőrzéséhez három feltételt használunk fel, nézzük meg ezeket:

- Ha egyik operátornak sincs olyan hatása, amely a *jelenlegicelok*ban található logikai mondat része lenne, akkor nem létezik ésszerű lépés.

- Ha a *jelenlegicelok* regresszálása *C*-t határozatlanná teszi, akkor bármely terv amely a *CS* cselekvést hozzáadja ezen a ponton az *utvonal*hoz, az a terv működésképtelen lesz. Már említettük, hogy a regresszió azokat a leggyengébb előfeltételeket adja vissza, amelyeknek a *CS* cselekvés végrehajtása előtt igazaknak kell lenniük, hogy később majd a *jelenlegicelok* elemei igazak legyenek a végrehajtás után. Ha egy *CS* cselekvés hatásai konfliktusba kerülnek a *jelenlegicelok*kal, akkor a leggyengébb előfeltétel határozatlan

lenne, hiszen teljesen mindegy, hogy mi volt igaz *CS* előtt, a terv nem működne. Jó példa erre, amikor a *(rajta A B)* ∧ *(rajta B C)* mondatot akarjuk regresszálni a 'tegyük A-t a B-ről az Asztalra' operátorral. Ez negálja a *(rajta A B)*-t és ebből következik, hogy a leggyengébb előfeltételek meghatározatlanok.

– Ha $C \supset$ *jelenlegicelok*, akkor nincs értelme a *CS* cselekvést hozzáadni az *utvonal*hoz ugyanazon indok miatt, mint amit az első pontban megfogalmaztunk.

## 4.3. Értékelés

Az algoritmusokkal kapcsolatban két kérdés merülhet fel. Az első, hogy ha kapunk egy tervet, akkor az tényleg működik-e. A második pedig, hogy ha a terv létezik, akkor van-e olyan nem determinisztikus választás-sorozat, amely ezt megtalálja - ez a teljesség kérdésköre. Mindkét eljárás esetén igen válaszokat adhatunk a kérdésekre. A lényegesebb kérdés az, hogy melyik algoritmus a gyorsabb.

Nem determinisztikus formájukban a komplexitásuk megegyezik: tökéletes szerencsével mindkét eljárás ugyanannyi - legyen $n$ darab - döntést hoz a megoldás megtalálása előtt. Azonban valós implementáció mellett a nemdeterminisztikusságot kereséssel kell megoldani, így az igazi kérdés valójában az, hogy hány választási lehetőséget kell figyelembe venni - nevezzük ezt $b$ darabnak. Már egy kis eltérés b mennyiségében nagy különbségekhez vezethet a tervezés hatásosságát figyelembe véve, hiszen a „nyers-erő" módszer keresési időigénye: $b^n$.

Ha feltételezzük, hogy a tervezési feladatban a célfeltételben csak kis számú literált használunk fel, akkor a regressziós tervkészítés elágazási tényezője feltehetően kisebb lesz minden egyes választásnál és ennek eredményeképp gyorsabb lesz a futási ideje is. Hogy ezt belássuk, elég azt megnézni, hogy hány cselekvést tudunk elvégezni a kezdőállapotban. Ehhez képest csak pár olyan cselekvés van, ami ténylegesen közelebb visz az elérendő célhoz. Mivel a progresszív keresésnek minden olyan cselekvést figyelembe kell vennie, amelyek előfeltételei kielégítik a kezdőállapotot, ezért nem fog profitálni a tervező célvezérelt útmutatásaiból.

Ezek mellett léteznek még más keresési módszerek is - például Kétirányú keresés vagy Közepek-Végek Analízis (Means-Ends Analysis). Utóbbi stratégiát használta a General Problem Solver rendszer is, de a GPS-szerű rendszerek nem teljesek - például nem tudják megoldani a Sussman Anomália problémát.

# 5. Tervkészítés tervtérben való kereséssel

Először egy Noah nevű tervező (Earl Sacerdoti, 1974) épült erre a megközelítésre. A rendszer az állapottér helyett a tervtérben keres, ahol a csomópontok részlegesen definiált terveket, az élek

pedig tervfinomító műveleteket jelentenek, mint például egy cselekvés hozzáadása a tervhez. A tervező ebben az esetben nem útvonalat ad vissza, a célállapot magát a megoldást jelenti. Ha maradunk az eddigi példánknál, akkor a kezdőállapot egy üres terv, míg a célállapot egy működő terv a Sussman Anomália feladathoz.

## 5.1. Teljesen rendezett tervkészítés

Ha feltételezzük, hogy egy terv teljesen rendezett cselekvéssorozat, akkor a RegKereses algoritmus valójában hasonló egy tervtér keresőhöz. Minden rekurzív hívás után továbbadjuk az *utvonal* argumentumot, amely egy teljesen rendezett tevékenység-sorozat. A keresés természete tulajdonképpen nézőpont kérdése: ha a RegKereses eljárást úgy tekintjük, hogy az állapottérben keres, akkor egy regressziós tervező, ha viszont teljesen rendezett tervek közötti keresőként szemléljük, akkor pedig tervfinomító operátorokkal módosítjuk a tervet az új cselekvéseknek az akciósorozat elejéhez történő hozzáadásával.

Tervtérben való keresésben gondolkodni azért előnyös, mert segít alternatív tervfinomító műveleteket készíteni és ez jobb tervező algoritmusokat eredményez. Ezeket az algoritmusokat itt nem taglaljuk, helyette inkább nézzünk meg egy jobb módszert, amelyhez más tervreprezentációs módszert kell választanunk.

## 5.2. Részben rendezett tervkészítés

A legkisebb elkötelezettség tervezési technika lehetővé teszi, hogy a tervezést rugalmasan kezeljük és egyes döntéseket későbbre halaszthatunk. Ahelyett, hogy korai döntések alapján egy teljesen rendezett tervet állítunk elő, a terv ebben az esetben egy részben rendezett cselekvéssorozat - csak a nélkülözhetetlen döntéseket hozzuk meg.

Jó példa erre egy külföldi utazás megszervezése. Az utazáshoz meg kell vennünk a repülőjegyeket, de szükség van még valamilyen turisztikai könyvre is. Egy meghatározott időpillanatig teljesen mindegy, hogy melyik cselekvést hajtjuk végre előbb.

Egy terv ábrázolása a következő elemhármassal történik: $\langle CS, M, K \rangle$. A $CS$ a cselekvések halmazát jelöli, az $M$ a rendezési megszorításokat tartalmazza, a $K$ pedig okozati kapcsolatok együttese. Ha $CS = \{CS_1, CS_2, CS_3\}$, akkor $M$ egy lehetséges halmaza: $\{CS_1 < CS_2, CS_2 < < CS_3\}$. Ezek a megszorítások egy olyan tervet határoznak meg, ahol a $CS_3$ az utolsó a cselekvések között, de nem mondják meg, hogy melyik lesz a legelső. A rendezési megszorítások konzisztensek, hiszen létezik legalább egy teljesen rendezés, amely eleget tesz $M$-nek. Mivel a legkisebb elkötelezettség módszerét használó tervezők tervfinomítással dolgoznak, ezért korlátozás kielégítést kell alkalmazniuk, hogy megőrizzék $M$ egységességét.

A fő szempont ennél a módszernél a múltbéli döntések nyomon követése és a döntések okainak nyilvántartása. Ha például megvettük a turisztikai könyvet, de indulás közben egy másik cél miatt le kell raknunk egy időre az asztalra, akkor a későbbiekben emlékeznünk kell rá, hogy újra felvegyük. Ezért explicit módon rögzítenünk kell a cselekvések közötti függőségeket. Esetünkben a K halmaz jelenti az okozati kapcsolatok adatstruktúráját, amit Austin Tate talált fel és használta először a NONLIN nevű tervezőjében 1977-ben.

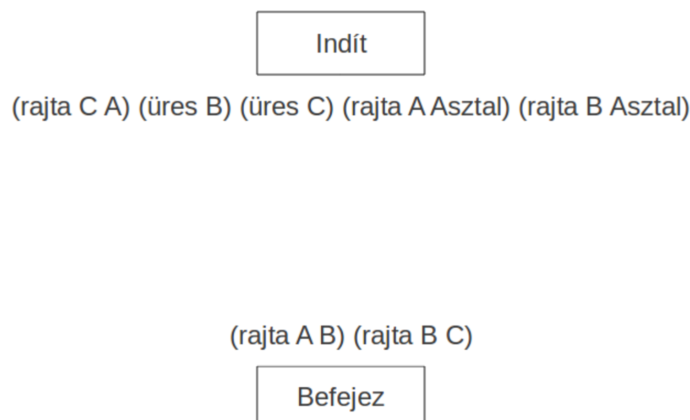
A struktúra három elemből áll és a következőképpen jelöljük: $CS_{gy} \xrightarrow{Q} CS_f$. Van két cselekvés mutatónk: egy gyártó és egy fogyasztó $(CS_{gy}, CS_f)$ és közöttük egy $Q$-val jelölt elem, amely $CS_{gy}$ hatása és egyben $CS_f$ előfeltétele is. Az okozati kapcsolatok arra valók, hogy azonosítsuk, ha egy új cselekvés hatása ütközik egy régebben meghozott döntésbe - ezt veszélynek nevezzük. Ha egy harmadik $CS_1$ cselekvés veszélyezteti $CS_{gy} \xrightarrow{Q} CS_f$-et, akkor a következők két feltétel teljesül:

$$M \cup \{CS_{gy} < CS_1 < CS_f\} \ \textit{konsziztens és}\ CS_1\ \textit{hatása}\ \neg Q$$

Például, ha a terv tartalmazza $CS_{gy} \xrightarrow{Q} CS_f$-et és $CS_{gy}$ állítja, hogy $Q$ = *(rajta A B)*, ami a $CS_f$ előfeltétele, akkor $CS_1$ veszélyként értelmezhető abban az esetben, ha levenné az $A$ kockát a $B$-ről. Ilyen esetben a tervező algoritmusnak megelőző lépéseket kell tennie. Egyik lehetőség, hogy rendezési megszorítást ad hozzá a tervhez, hogy biztosítva legyen $CS_1$ végrehajtása $CS_{gy}$ előtt - ezt lefokozásnak hívjuk, az ellentettjét pedig előléptetésnek.

**Null tervek**

Egy tervtér keresőt legegyszerűbben egységes reprezentációval írhatunk le, tehát ugyanazzal a módszerrel kell definiálnunk a tervezési problémákat és a hiányos terveket is. Ahhoz, hogy ezt megvalósítsuk a kezdeti állapot leírását és a célfeltételeket egy null tervben kell egyesíteni.




2. ábra: A Sussman Anomália null terve

A null terv két cselekvést - $\{CS_0, CS_\infty\}$ -, egy megszorítást - $\{CS_0 < CS_\infty\}$ tartalmaz és nincsenek okozati kapcsolatok. $CS_0$ az Indít elnevezésű kezdőcselekvés, aminek nincsenek előfeltételei és a hatása definiálja, hogy mi fog teljesülni a kezdőállapotban. A $CS_\infty$ a Befejez cselekvést szimbolizálja, aminek nincs hatása, de előfeltétele a tervezési feladat céljainak konjunkciója.

### 5.2.1. A POP algoritmus

A POP egy regresszív részben rendezett tervező. Az eljárás első meghívásakor a tervet reprezentáló elemhármasnak a probléma null tervét, a *feladatok*nak a célok konjunkciójának listáját kell megadnunk. A végső terv rendezési megszorításai részben rendezett tervet is specifikálhatnak, ebben az esetben bármely $M$-mel konzisztens teljesen rendezett terv garantáltan megoldása a tervezési feladatnak.

POP($\langle CS, M, K\rangle$,*feladatok,CS*):

Megállási feltétel: **Ha** a *feladatok* üres **akkor return** $\langle CS, M, K\rangle$.

Célkiválasztás: Legyen $\langle Q, CS_{szukseges}\rangle$ egy pár a *feladatok* halmazból.

Cselekvés kiválasztása: Legyen $CS_{hozaad}$ egy olyan választott cselekvés
amelynek hatása $Q$. Ez lehet egy új példányosított cselekvés $CS$-ből
vagy egy $CS$-ben meglévő is, amely konzisztensen rendezhező
$CS_{szukseges}$ elé. **Ha** ilyen nincs **akkor return** kudarc.
Legyen $K' = K \cup \{CS_{hozzaad} \xrightarrow{Q} CS_{szukseges}\}$ **és**
$M' = M \cup \{CS_{hozzaad} < CS_{szukseges}\}$.
**Ha** $CS_{hozaad}$ újonnan példányosított cselekvés **akkor**
$CS' = CS \cup CS_{hozzaad}$ **és** $O' = O' \cup \{CS_0 < CS_{hozzaad} < CS_\infty\}$
**különben** $CS' = CS$.

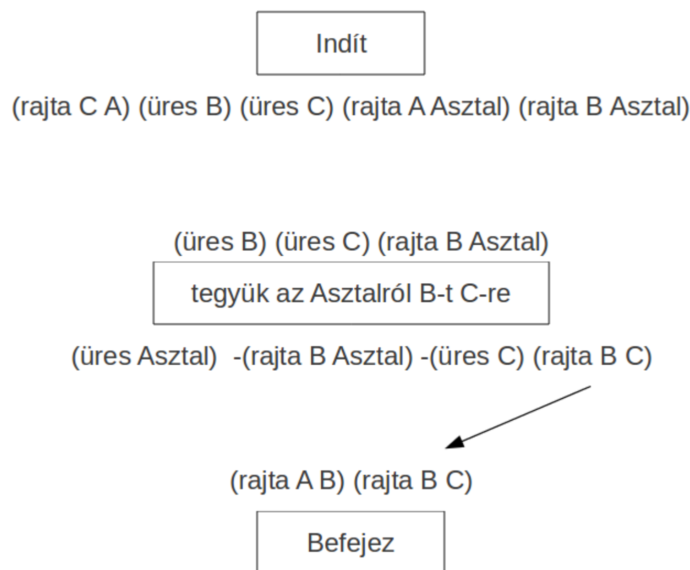
Célhalmaz frissítése: $feladatok' = feladatok - \{\langle Q, CS_{szukseges}\rangle\}$
**Ha** $CS_{hozaad}$ újonnan példányosított **akkor**
minden egyes előfeltételben található konjunkt - $Q_i$ - esetén
adjuk $\langle Q_i, CS_{hozzaad}\rangle$-ot a $feladatok'$-höz.

Okozati kapcsolatok védelme: Minden olyan $A_t$ cselekvés esetén amely
veszélyezteti a $\{CS_{gy} \xrightarrow{R} CS_f\} \in K'$ okozati kapcsolatot,
választani kell egy konzisztens rendezési megszorítást.
Ez vagy lefokozás (adjuk hozzá $A_t < A_{gy}$-t $M'$-höz)
vagy előléptetés (adjuk hozzá $A_f < A_t$-t $M'$-höz) lehet.
Ha egyik megszorítás se konzisztens **akkor return** kudarc.

Rekurzív hívás: POP($\langle CS', M', K'\rangle$, $feladatok'$,*CS*)

Az algoritmus bemutatásához a jól ismert kockavilág problémát vesszük elő. A *feladatok*at $\langle Q, CS_i \rangle$ elempárok alkotják, ahol $Q$ a $CS_i$ előfeltételének a konjunktja. Az eljárás meghívásakor a *feladatok*ban két elem található: $\langle$ *(rajta A B)*, $CS_\infty \rangle$ és $\langle$ *(rajta B C)*, $CS_\infty \rangle$. Az algoritmusnak választania kell a két részcél közül, de nem adtunk meg 'választ' metódust. Ennek az az oka, hogy mindegy, hogy melyiket választjuk, amíg a teljességet szem előtt tartjuk. Mindkét döntést meg kell majd hoznunk, tehát nincs értelme egy keresésen alapuló megoldásnak ebben az esetben. Ez nem jelenti azt, hogy a döntés lényegtelen, mert elképzelhető, hogy egy választás gyors megoldást eredményez és a különböző részcélok érveléseinek összevetése is hasznos lehet. A végeredményben viszont egy nem determinisztikus algoritmus számára ez nem fontos, hiszen mindkét választás esetén ugyanannyi döntést kell meghozni.

Válasszuk ki a *(rajta B C)*-t első részcélnak - $CS_{szukseges}$ legyen $CS_\infty$-re állítva. Most egy nem determinisztikus 'választ' metódus segítségével kiválasztunk egy olyan $CS_{hozaad}$ cselekvést amelynek *(rajta B C)* a hatása.

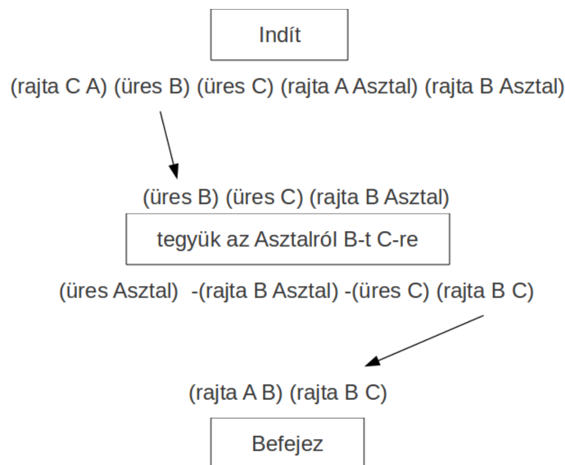



3. ábra: A 'tegyük az asztalról B-t C-re' cselekvést kiválasztva egy új okozati kapcsolattal is bővül a tervünk

Ha a 'tegyük az asztalról B-t C-re' esik a választás akkor a $CS_{hozzaad} \xrightarrow{(rajtaBC)} CS_\infty$ okozati kapcsolat lesz az $K'$-höz hozzáadva. Eközben a *feladatok* halmaza is frissül és a elemei a következők lesznek: *{(üres B) (üres C) (rajta B Asztal) (rajta A B)}*. Mivel semmilyen veszélyhelyzet nem áll fent, ezért tovább ugrunk a rekurzív híváshoz.

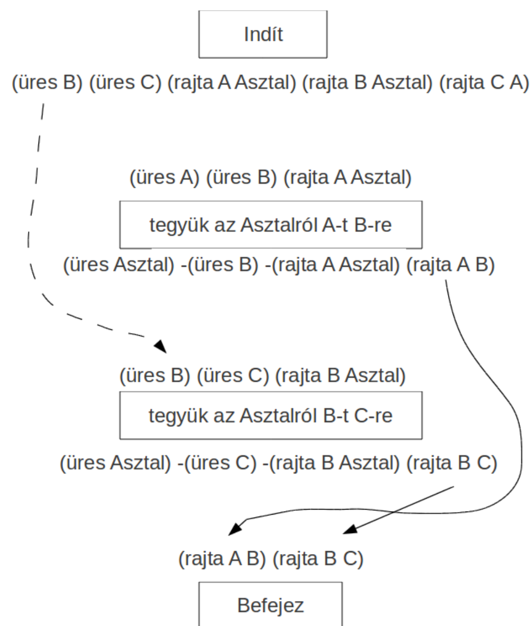
A kövezkető ciklusban a *feladatok* még nem üres, ezért tegyük fel, hogy az előző cselekvés előfeltételét, *(üres B)*-t választjuk ki $Q$-nak. A 'választ' metódus ezután nem determinisztikusan hoz egy döntést, legyen ez most egy olyan cselekvés, amelyet a tervező már egyszer felhasznált:

a $CS_0$ kezdőcselekvés. Ez a lépés egy okozati kapcsolatot ad az $K$-hoz és a *feladatok* elemei a következőkre módosulnak: *{(üres C) (rajta B Asztal) (rajta A B)}*.



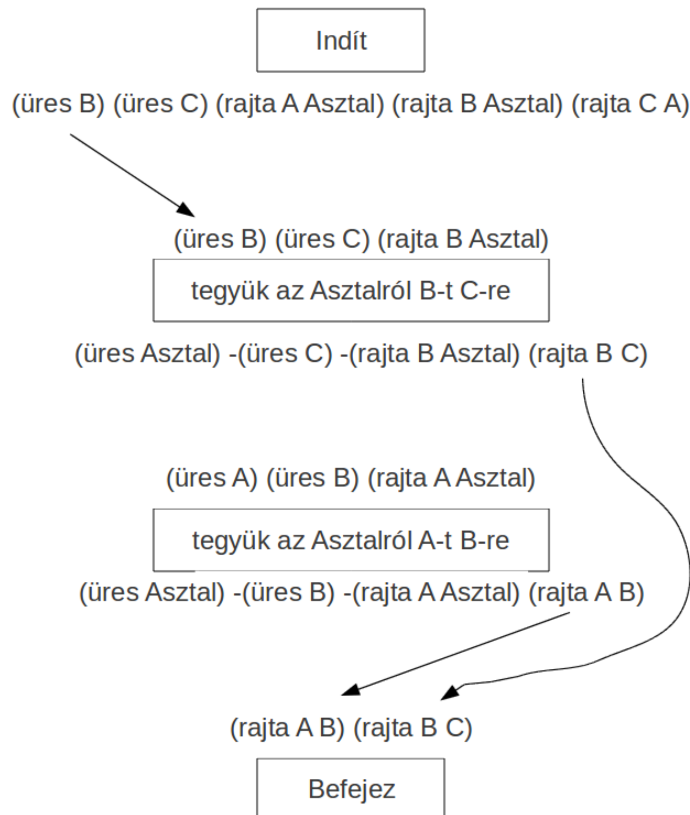

4. ábra: A tervező által már felhasznált cselekvést kiválasztva csökken a *feladatok* listája

A harmadik híváskor tegyük fel, hogy a *(rajta A B)* cél kerül kiválasztásra, valamint, hogy a $CS_{hozzaad}$ a 'tegyük az asztalról A-t B-re' cselekvés lesz. Egy új korlátozási megszorítás lesz $K$-hoz hozzáadva: a cselekvéshez tartozik egy megszorítás, hogy $CS_\infty$ előtt kell lennie. Az okozati kapcsolatok felülvizsgálatához érkezve azt vehetjük észre, hogy a 'tegyük az asztalról B-t C-re' és a 'tegyük az asztalról A-t B-re' cselekvésekre az a megszorítás érvényes, hogy meg kell előzniük $CS_\infty$-t, de $M$ nem tartalmaz megszorítást a két cselekvés sorrendjéről és az előbbi akció ráadásul negálja *(üres B)*-t.



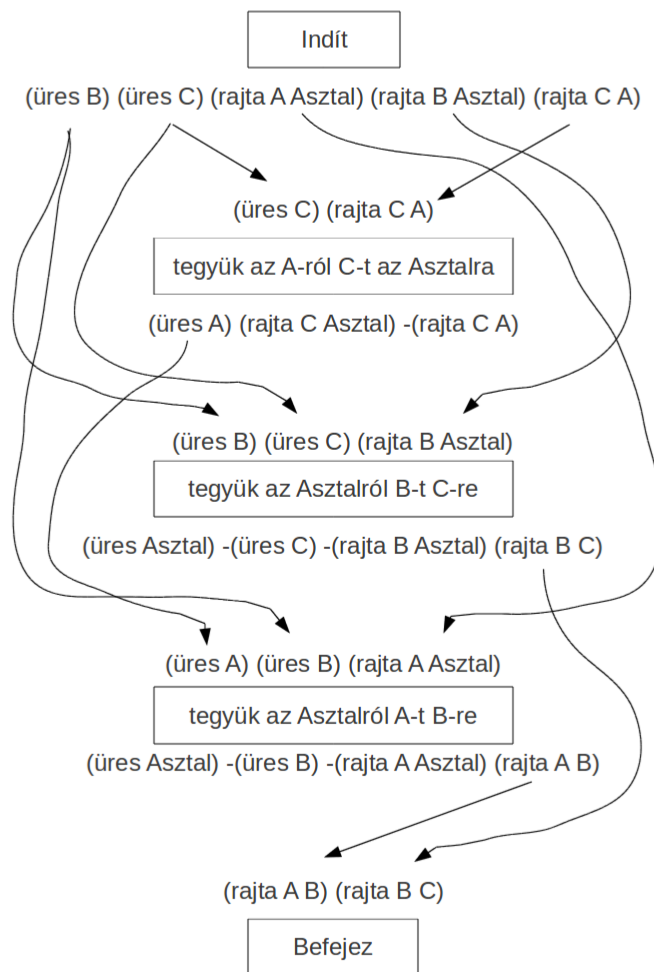

5. ábra: A 'tegyük az asztalról A-t B-re' cselekvés veszélyeztet egy meglévő okozati kapcsolatot

Ez veszélyezteti a $CS_0$-ból kiinduló *(üres B)* kapcsolatot is, ezért az algoritmusnak nem determinisztikusan választania kell egy rendezési megszorítást. Két lehetséges megoldás létezik: vagy az $A$-t mozgató cselekvést kényszerítjük, a $B$-t mozgató cselekvés után vagy az $A$-t mozgató cselekvésnek meg kell előznie a $CS_0$ kezdőcselekvést. Mivel az algoritmus cselekvést kiválasztó része garantálja, hogy minden tevékenység a kezdőcselekvés után kerüljön végrehajtásra, ezért az utóbbi választás inkonzisztenssé tenné $M$-met.




6. ábra: A terv állapota az előléptetés után

A POP a veszélyeztető cselekvést előlépteti az okozati kapcsolat fogyasztója után. Az eljárás további működése az eddigiekkel megegyező módon történik, ezért inkább nézzük meg, hogy milyen előnyt nyújt ez a megközelítés az állapottér-kereséssel szemben.




7. ábra: A Sussman Anomália megoldása

## 5.3. Értékelés

Egy keresési algoritmus költsége $fl^d$. A képletben a $f$ az az időmennyiség, amelyet az állapottérben egy csomópont feldolgozása igényel; a $l$ az elágazási tényező; az $d$ pedig a megoldás eléréséig végrehajtott nem determinisztikus döntések száma.

Az $f$ időmennyiség a RegKereses és a POP algoritmusokat összehasonlítva csak kis mértékben különböznek, ezért ezzel nem érdemes foglalkozni mélyre hatóbban.

A nem determinisztikus 'választ' metódus hívások száma változó lehet, de ez sem sugallja azt, hogy a POP algoritmus gyorsabb lenne. A RegKereses algoritmus egyszer hívja meg a 'választ' metódust cselekvésenként, a POP pedig minden előfeltételben szereplő konjunktnál. Ha az okozati kapcsolatok veszélyeztetését is figyelembe vesszük akkor kijelenthetjük, hogy a legtöbb esetben a POP algoritmus nagyobb $d$-t eredményez.

A képletben a meghatározó paraméter a $l$. A POP eljárásban a cselekvés kiválasztásakor csak azokat a tevékenységeket kell figyelembe venni, amelyeknek a hatása az éppen aktuális

részcél - $Q$. A RegKereses-ben a 'választ' metódusnak minden olyan cselekvést számításba kell vennie, amelyeknek hatása a *jelenlegicelok* bármely eleméhez kötődik. A legkisebb elkötelezettség módszerével és a rendezési megszorításokkal az elágazási tényezőt a *feladatok* vagy a *jelenlegicelok* átlagos méretével tudjuk csökkenteni és ez teszi hatékonyabbá a POP algoritmust az állapottér-keresőknél.

# 6. Tervkészítés nem determinisztikus problémakörben

Az eddigiekben csak klasszikus tervkészítési feladatokkal foglalkoztunk, amelyek teljesen megfigyelhetőek, végesek és statikusak, továbbá a cselekvések hatása is determinisztikus. Ilyen esetekben egy terv végrehajtása mindig megjósolható eredményt ad, hiszen nem fordulhatnak elő váratlan történések. Azonban, ha a probléma világa bizonytalan, akkor a tervezőnek követnie kell az eseményeket a terv megvalósítása közben és váratlan helyzetek esetén módosítania kell a tervet.

A tervezőnek figyelembe kell vennie a hiányos és hibás információkat is. A probléma világa ettől még helyes marad, csak a róla mintázott modell nem lesz teljes. A hiányosság a nemdeterminisztikusságból és a bizonytalan megfigyelhetőségből is adódhat, például egy kapu vagy nyitva van vagy zárva, de az is előfordulhat, hogy a világ nem felel meg a modellünknek. Nem biztos, hogy egy kulcs például kinyitja azt a kaput, de ettől még hihetem ezt. Ha a tervező nincs kellőképpen felkészítve a pontatlan adatok kezelésére, akkor ez a legtöbbször eredménytelen tervkészítéshez vezet.

A hiányzó ismeretek feltárásának lehetőségét a világ nemdeterminisztikussága határozza meg. Korlátos nemdeterminisztikusság esetén a cselekvések kimenetele bizonytalan, de a következményeket megadhatjuk a cselekvés leírásában. Például egy pénzérme feldobása előtt már biztosan tudhatjuk, hogy két lehetséges eredmény következhet be: vagy fejet, vagy írást kapunk. Egy algoritmus ezt úgy kezelheti, hogy a tervet felkészíti az összes eshetőségre. Nem korlátos nemdeterminisztikusság esetén az előfeltételek és következmények halmaza ismeretlen vagy túl nagy ahhoz, hogy felsoroljuk őket. Ebben az esetben az algoritmust úgy kell megtervezni, hogy módosíthassa a terveit és adatbázisát egyaránt.

Korlátos nemdeterminisztikusság esetén két tervkészítési módszer használható:

– Érzékelőmentes (alkalmazkodó) tervkészítés: Az érzékelőmentes módszer általános terveket készít és biztosítja, hogy a célt minden körülmény esetén elérjük. Kényszerítéssel dolgozik, ami azt feltételezi, hogy a világ akármilyen állapotba átvihető. Ez néha lehetetlen, ezért egyes problémák esetén nem alkalmazható az algoritmus.

– Feltételes (eshetőségi) tervkészítés: A feltételes algoritmus egy olyan tervet készít, ahol minden egyes következményhez más ágak tartoznak. A tervező érzékeli a feltételeket speciális cselekvések segítségével és a megfelelő ágon megy tovább.

Nem korlátos nemdeterminisztikusság esetén is két módszer alkalmazható:

– Végrehajtás monitorozás és újratervezés: használhat klasszikus, érzékelőmentes és feltételes algoritmusokat, továbbá minden lépésben dönt, hogy az aktuális terv használható-e. Ha valami hiba történik, akkor automatikusan újratervezünk, ezzel kezelve a nem korlátos nemdeterminisztikusságot.

– Folytonos tervkészítés: képes kezelni bármilyen nem várt eseményt, például új célok megjelenését vagy akár célok eldobását is a célújraformálás segítségével.

A tervkészítők közötti különbségek bemutatása végett definiáljunk egy egyszerű problémát: legyen egy asztalunk, egy székünk és pár darab festékes dobozunk. Célunk, hogy az asztal és a szék színe azonos legyen úgy, hogy semelyik tárgynak sem ismerjük a színét.

– Egy klasszikus tervkészítő algoritmus nem tud mit kezdeni a problémával, mert a kezdőállapot hiányosan van specifikálva (a tárgyak színe nincs meghatározva).

– Egy érzékelőmentes tervkészítő által készített terv a kényszerítésen alapul. A festékesdobozok közül választunk egyet és a festékkel az asztalt és a széket is befestjük. A kényszerítés helytálló, ha az információk megszerzése költséges vagy nem lehetséges. A való életben például az orvosok gyakran egy általános hatású gyógyszert ajánlanak, hiszen csak egy komplett kivizsgálás után lehetne egy konkrét betegségre szánt gyógyszert ajánlani.

– Egy feltételes tervkészítő az érzékelőmentes módszernél hatékonyabban dolgozik. Ha az asztal és a szék színe már megegyezik, akkor megvan a megoldás. Ha a festékesdobozok között talál olyan színűt, amely megegyezik az egyik tárgy színével, akkor a másikat befesti vele. Ha ez sem teljesül, akkor pedig mindkét tárgyat befesti egy véletlenszerűen választott színnel.

– Egy újratervező tervkészítő a feltételes algoritmussal megegyező tervet állít elő, de egy ennél kevesebb ággal rendelkezőt is generálhat, amelyet majd a végrehajtás alatt fog kibővíteni, ha szükséges. Ha a 'fessük le az Asztalt Sárgára' cselekvésnek a következménye (Asztal Sárga), akkor egy feltételes tervkészítő vélheti úgy, hogy a következmény teljesül a cselekvés végrehajtása után, de ezt az újratervező algoritmus felülbírálhatja bizonyos esetekben.

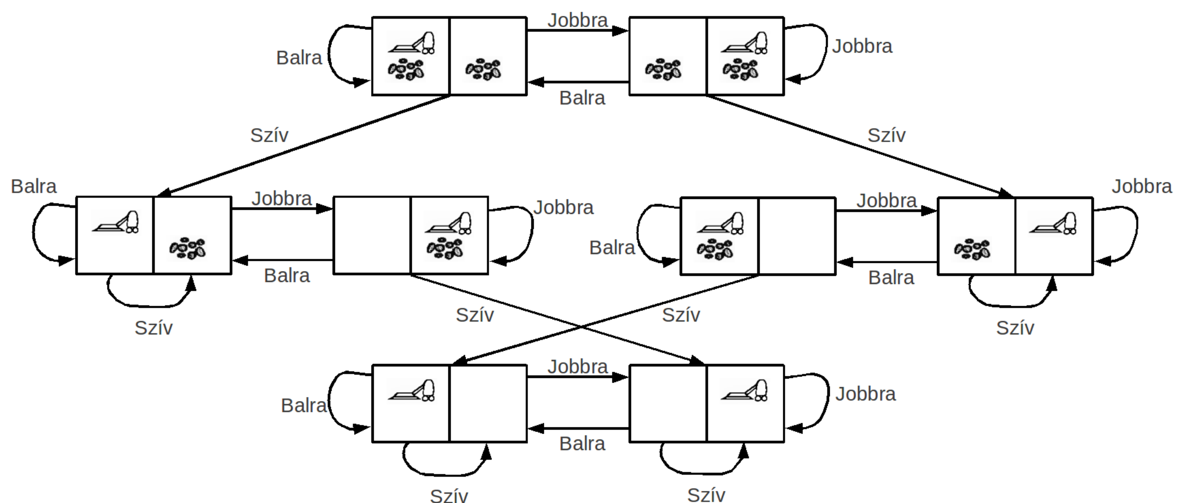
– Egy folytonos tervkészítő is az eddigiekhez hasonló tervet állít elő, de egy új cél megadása esetén előfordulhat, hogy egy régebbi célt el kell halasztanunk.

A gyakorlatban a tervkészítők ezen módszereknek a kombinációival dolgoznak. Általánosságban elmondható róluk, hogy csak olyan esetekre készítenek feltételes tervet, amelyek lényeges következményekkel járnak és jelentős esély van arra, hogy ezek bekövetkezzenek. Például egy hosszú autóút esetén tekintettel kell lenni a műszaki problémákra is, de egy rövid táv során sokkal kevesebb a meghibásodás esélye. Fontos megjegyezni még, hogy ezek a tervkészítők alkalmatlanok egy terv költsége és sikeressége közötti összefüggések figyelembe vételére.

## 6.1. Feltételes tervkészítés teljesen megfigyelhető világban

A feltételes tervkészítés alapvető jellemzője, hogy a terv bizonyos pontjain ellenőrzi a világban bekövetkező eseményeket, ezzel biztosítva a bizonytalanságkezelést. A könyebb érthetőség miatt először egy teljesen megfigyelhető világban nézzük meg egy ilyen tervező működését.

Egy teljesen megfigyelhető világban a tervkészítő minden esetben teljes ismeretekkel rendelkezik az aktuális állapotról, nem determinisztikus világ esetén viszont a tervkészítő nem tudja előrejelezni a cselekvések kimenetelét. Egy feltételes tervkészítő a tervezés közben feltételes lépéseket illeszt be a tervbe, amelyek ellenőrzik végrehajtás közben a világ állapotát és ha szükséges, változtatásokat eszközölhet. A példánkhoz először nézzük meg a porszívóvilágot.




8. ábra: A porszívóvilág állapottere

A fenti ábrán jól látszik, hogy háromféle cselekvésünk van: *Balra*, *Jobbra* és *Szív*. Ahhoz, hogy STRIPS segítségével reprezentáljunk egy nem determinisztikus feladatot, alkalmaznunk

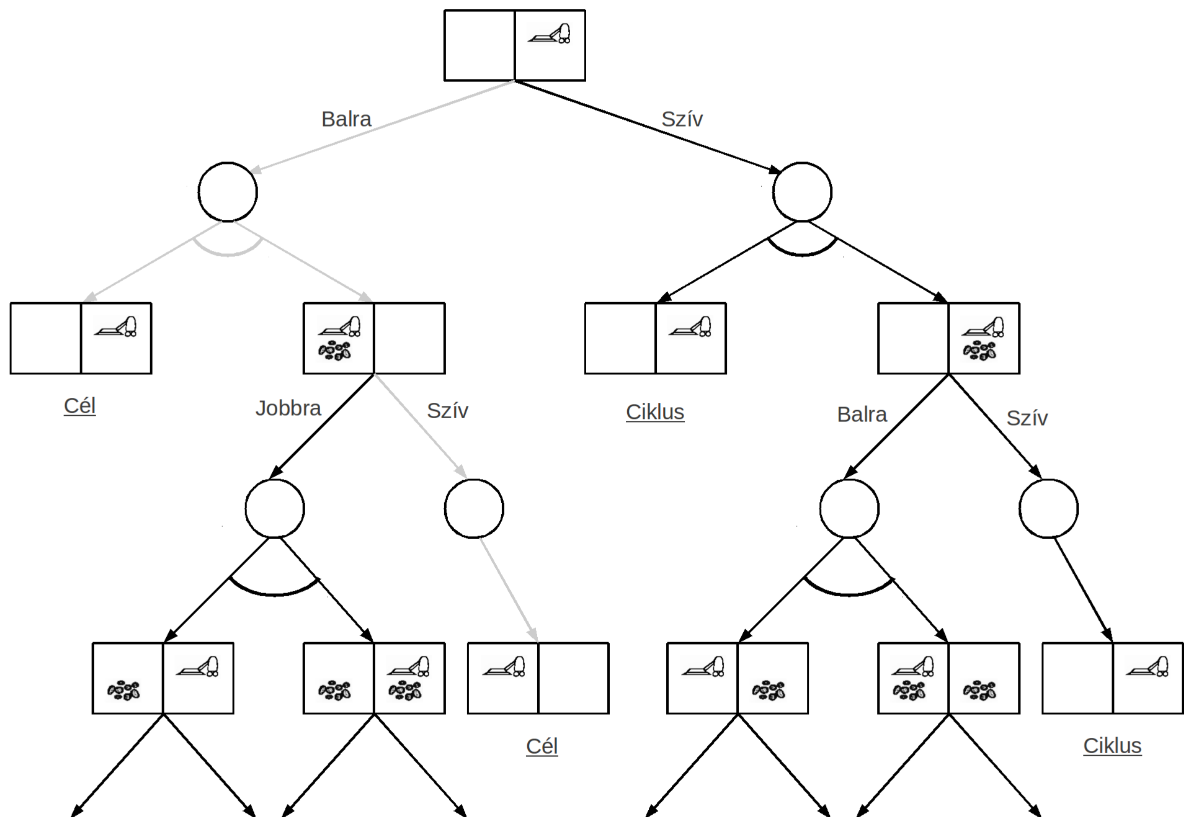
kell diszjunktív és feltételes következményeket is. A diszjunktív következmény megengedi, hogy egy cselekvésnek több hatása is legyen. Ha a *Balra* cselekvés néha eredménytelen, akkor az alábbi módon írhatjuk ezt le:

*Cselekvés(Balra,*
*Előfeltétel: OttJobb,*
*Következmény: OttBal* $\vee$ *OttJobb)*

A feltételes következmények lehetővé teszik, hogy egy cselekvés hatása függjön attól az állapottól, amelyben elvégezzük. A Szív cselekvés a következőképpen definiálható:

*Cselekvés(Szív,*
*Előfeltétel: ,*
*Következmény: (**ha** OttBal: TisztaBal)* $\wedge$ *(**ha** OttJobb: TisztaJobb))*

Ezek mellett még szükségünk van feltételes lépésekre is, melyek összeillesztésével a tervek fát alkotnak - például a ***ha** OttBal* $\wedge$ *TisztaBal* ***akkor*** *Jobbra* ***különben*** *Szív* egy ilyen lépést ír elő. Konkrét példaként nézzük meg a dupla-Murphy porszívóvilág problémát.




9. ábra: A dupla-Murphy probléma fájának első két szintje és megoldása szürke vonallal jelölve

A kezdőállapotban mindkét oldal tiszta és a porszívó a jobb oldalon van. A cselevéseket egy állapot csomópontján hajtjuk végre, a körrel jelölt valószínűségi pontokon pedig a természettől függ a cselekvés eredménye. A feladat megoldása egy részfa, ami jelen esetben a következő tervnek felel meg:

*[Balra, **ha** OttBal ∧ TisztaBal ∧ TisztaJobb **akkor** [ ] **különben** Szív]*

A feltételes terveknek a cselekvések kimenetelétől függetlenül kell működniük. Ilyen problémával találkozhatunk a kétszemélyes játékokban is, ahol olyan cselekvéseket kell választanunk, amelyek garantálják a győzelmet az ellenfél döntéseitől függetlenül. A nem determinisztikus tervkészítési feladatokat emiatt természet elleni játékoknak is szokták nevezni. A játékok megoldásának általános módszere a minimax algoritmus, de a feltételes tervkészítéshez ennek egy módosított változatát használjuk. A MAX és a MIN csomópontok helyett VAGY és ÉS csomópontok lesznek, valamint nem egy lépést, hanem egy feltételes tervet kell az algoritmusnak előállítani. A VAGY csomópontokban a terv egy cselekvés, míg az ÉS csomópontokban ***ha*** ... ***akkor*** ... ***különben*** ... tevékenységek sorozata mindegyik feltételezhető kimenetelhez. A nem determinisztikus környezetek által előállított ÉS-VAGY gráfok keresésére szolgáló algoritmus a következő:

*EsVagyGrafKereses(feladat)*
  VagyKereses(KiinduloAllapot[feladat], feladat, [ ])

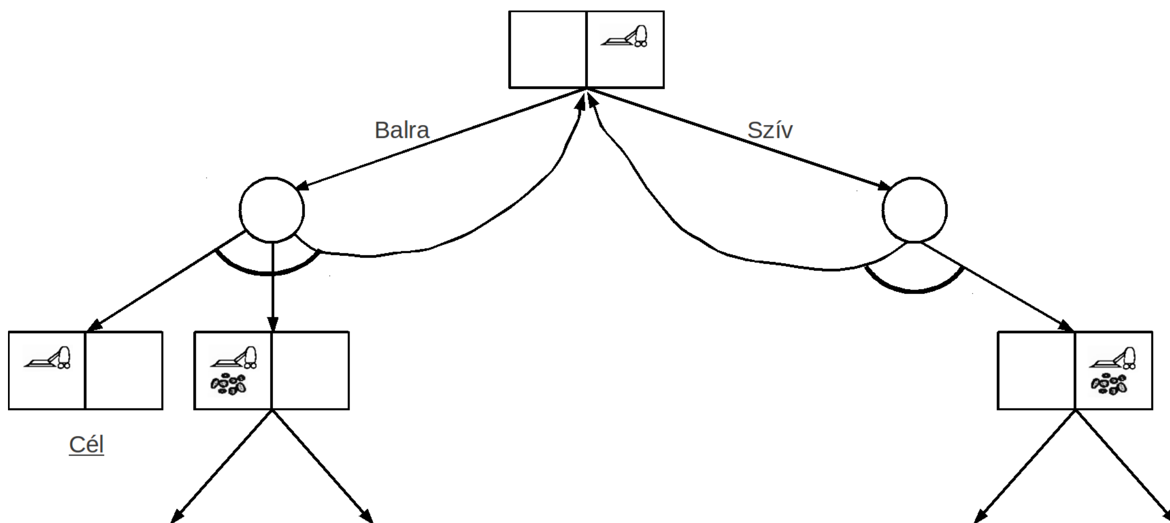
*VagyKereses(allapot, [feladat], feladat, [ ])*
  **Ha** CelTeszt[*feladat*](*allapot*) **akkor return** üres terv
  **Ha** *allapot* az *utvonal*on **akkor return** kudarc
  **Minden egyes** *cselekves*, *allapothalmaz*-**ra a** Kovetoallapotok[*feladat*](*allapot*)-**ban**
    *terv* = EsKereses(*allapothalmaz*, *feladat*, [*allapot*|*utvonal*])
    **Ha** *terv* ≠ kudarc **akkor return** [*cselekves*|*terv*]
  **return** kudarc

*EsKereses(allapothalmaz, feladat, terv)*
  **Minden egyes** $s_i$-**re az** *allapothalmaz*-**ban**
    *terv* = VagyKereses($s_i$, *feladat*, *terv*)
    **Ha** *terv* = kudarc **akkor return** kudarc
  **return** [**Ha** $s_1$ **akkor** $terv_1$ **különben ha** $s_2$ **akkor** $terv_2$ **különben** ...
      **ha** $n_{-1}$ **akkor** $terv_{n-1}$ **különben** $terv_n$]

Érdemes megnézni az algoritmus cikluskezelését, hiszen előfordulhat, hogy egy cselekvésnek nincs vagy helytelen a következménye - ezek javítható hibák. Ha az aktuális állapot megegyezik a gyökértől a jelen állapotig vezető út egy állapotával, akkor az eljárás kudarccal tér vissza. Ez azt jelenti, hogy létezik egy olyan ciklusmentes megoldás, amely hozzáférhető az aktuális állapot korábbi előfordulásából, tehát az állapot elhagyható. Így érjük el, hogy az eljárás minden véges állapottér esetén leálljon, viszont azt nem vizsgáljuk, hogy az aktuális állapot egy másik útvonalon lévő állapottal megegyezik-e.

Az algoritmus által előállított terv feltételes lépéseiben a teljes állapottér ellenőrzésre kerül, de a teszteket leredukálhatjuk egyváltozósra is abban az esetben, ha az adott állapot teljes egészében megfigyelhető. A példafeladatunk tervét emiatt leegyszerűsíthetjük a következőre: *[Balra, **ha** TisztaBal **akkor** [ ] **különben** Szív]*, hiszen a *TisztaBal* teszt garantálja az ÉS csomópont állapotainak két egyelemű halmazba sorolását.

Előfordulhatnak olyan nem determinisztikus problémák is, melyekben a cselekvések végrehajtása sikertelen is lehet, ezért többször kell próbálkoznunk. Ezek közé tartozik a tripla-Murphy porszívóvilág probléma is.




10. ábra: A tripla-Murphy probléma fájának első szintje

Ebben a feladatban az előző tervünk már nem biztos, hogy működne, mert az első szint után már nincsenek ciklus nélküli megoldások és a fent vázolt algoritmus kudarcot vallana. Az EsVagyGrafKereses eljárás kis módosítással ciklust tartalmazó terveket is képes előállítani. A példafeladatunk megoldása a

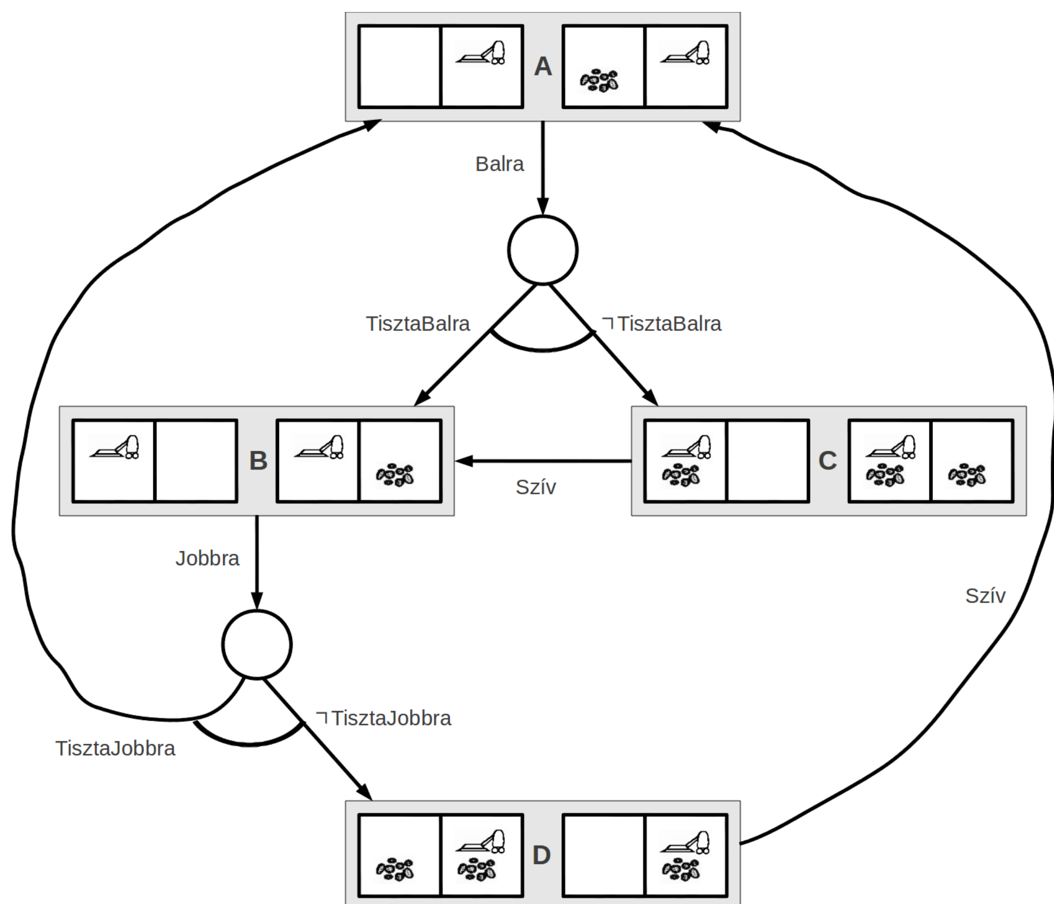
*[$C_1$ : Balra, **ha** OttJobb **akkor** $C_1$ **különben ha** TisztaBal **akkor** [ ] **különben** Szív]*

terv, ahol egy $C_1$ címke segítségével addig próbáljuk végrehajtani a *Balra* cselekvést, amíg az eredményhez nem vezet. A ciklikus tervek nem hatékonyak annyira, mint a ciklus nélküliek, mert végtelen ciklusok is előfordulhatnak bennük, de ettől függetlenül érvényes megoldások.

## 6.2. Feltételes tervkészítés részlegesen megfigyelhető világban

Valós problémák esetén legtöbbször részlegesen megfigyelhető környezetek fordulnak elő. Egy ilyen világban a tervkészítő a kiinduló állapotról nem rendelkezik teljes információval, ezért a kezdőállapotot egy állapothalmazba való besorolással modellezhetjük - ezt hiedelmi állapotnak nevezzük.

A példafeladatunk legyen a váltakozó dupla-Murphy porszívóvilág probléma, amely abban különbözik az első feladattól, hogy a tervkészítő nem képes érzékelni a másik négyzetben lévő szemetet és a porszívó piszkot hagyhat maga után. A kezdőállapotban a porszívó a jobb oldali négyzeten van, ami tiszta.




11. ábra: A váltakozó dupla-Murphy porszívóvilág fájának egy részlete

A fenti gráfon az A hiedelmi állapot a kezdeti állapotunk. Ebben a helyzetben csak egy ésszerű cselekvést tudunk végrehajtani, ez pedig a *Balra*. Mivel a porszívó piszkot hagyhat maga után, ezért négyféle állapotot kaphatunk, amelyeket két külön hiedelmi állapotba csoportosíthatunk : a B-ben a *TisztaBal*, a C-ben a ¬*TisztaBal* információval rendelkezik a tervkészítő. A C állapotban a piszok eltakarítása a B állapotba vezet, ahonnan a *Jobbra* cselekvés szintén szemetet hagyhat maga után és ezért ismét négy kimenetel fordulhat elő. A tervkészítő ismereteitől függ, hogy visszatérünk-e az A állapotba - abban az esetben, ha *TisztaJobb* igaz - vagy a D-be kerülünk.

Látható, hogy az ilyen típusú feladatokban a hiedelmi állapotok ÉS-VAGY gráfot alkotnak, tehát az előző részben felvázolt algoritmus itt is használható feltételes tervek előállítására. Valójában a hiedelmi állapotot egy egységként tekintve mindig teljesen megfigyelhető, azaz a dupla-Murphy problémát ennek a feladatnak egy speciális eseteként foghatjuk fel, amelyben egy hiedelmi állapot egy tényleges állapotot tartalmaz.

A feladat teljes definiálásához először meg kell határoznunk az érzékelés módját, ami kétféleképpen működhet :

- Automatikus érzékelés : a tervkészítő mindegyik lépésben minden általa érzékelhető ismeretet megkap.
- Aktív érzékelés : az érzékelhető információk csak érzékelési cselekvésekkel érhetőek el.

A következő lépésben meg kell határozni a hiedelmi állapotok reprezentálási módját :

- Teljes állapotleírások : ez a legegyszerűbb definiálási forma, de hátránya, hogy költséges. Ha $n$ darab állítást tartalmaz egy állapot, akkor a hiedelmi állapot $2^n$ darab $n$ méretű állapotleírást foglalhat magában. Ha a tervkészítő csak az állítások kis részét ismeri, akkor a hiedelmi állapotok nagysága exponenciálisan növekszik. A példafeladatunk kezdőállapota teljes állapotleírással :

  $$\{(OttJobb \wedge TisztaJobb \wedge TisztaBal), (OttJobb \wedge TisztaJobb \wedge \neg TisztaBal)\}$$

- Logikai mondatok : egy hiedelmi állapotban lévő világok halmaza kifejezhető egyetlen logikai mondat segítségével is. Az általános logikai mondatok hátránya, hogy több különböző logikai mondat is definiálhatja ugyanazt az állapotot, ezért tételbizonyítást kell alkalmaznunk a tervkészítő algoritmusban. Ezt elkerülhetjük kanonikus reprezentáció használatával, ahol rendezett literálok konjunkciójával dolgozunk. A példafeladatunk kiinduló állapota tehát a következőképpen néz ki :

  $$OttJobb \wedge TisztaJobb$$

– Tudás ítéletállítások: a tervkészítő tudását írjuk le *K(P)* alakú állapotleírásokkal, ami azt jelenti, hogy az ágens tudja, hogy *P* igaz. Az állítások értelmezésénél zárt világ feltételezést használunk, tehát a nem meghatározott körülményeket hamisnak tekintjük. Ez azt jelenti, hogy az ítéletállításunk implicit módon tartalmazza a ¬*K(TisztaBal)* és a ¬*K(*¬*TisztaBal)* tényeket is. A kezdeti állapotunk az alábbira módosul:

$$K(OttJobb) \land K(TisztaJobb)$$

A feladatunkhoz a harmadik módszert fogjuk használni, mert az érzékeléssel kapcsolatos ismereteket hatékonyabban tudjuk kifejezni a segítségével.

Utolsó lépésként meg kell adnunk a cselekvéseket: példának nézzük meg a *Balra* lépés leírását.

*Cselekvés(Balra,*
*Előfeltétel: OttJobb,*
*Következmény: K(OttBal)* ∧ ¬*K(OttJobb)* ∧ ***amikor*** *TisztaJobb:* ¬*K(TisztaJobb)* ∧
∧ ***amikor*** *TisztaBal: K(TisztaBal)* ∧ ***amikor*** ¬*TisztaBal: K(*¬*TisztaBal))*

Az ágens a szabályoknak megfelelően piszkot hagyhat maga után: ismereti következményként ez azt eredményezi, hogy a tervkészítő a cselekvés végrehajtása után nem rendelkezik többé a *TisztaJobb* tudással. Láthatjuk azt is, hogy az előfeltételek és az „amikor” feltételek nem tudás állítások, hiszen ha a tervkészítő egy hiedelmi állapotban a *K(OttJobb)* ismerettel rendelkezik, akkor az az állítás biztosan igaz az adott állapotban. Ha egy állításról nincs információnk, akkor a hiedelmi állapotban lennie kell olyan világoknak melyekben az állapot igaz, és olyanoknak is ahol hamis - emiatt lesz egy cselekvés hatása több hiedelmi állapot is.

Aktív érzékelési módszer használata esetén csak érzékelési cselekvésekkel szerezhetünk új információkat a világról, példaként nézzük meg a *TisztasagEllenorzes* cselekvés leírását:

*Cselekvés(TisztasagEllenorzes,*
*Előfeltétel: ,*
*Következmény:* ***amikor*** *OttBal* ∧ *TisztaBal: K(TisztaBal)* ∧
***amikor*** *OttBal* ∧ ¬*TisztaBal: K(*¬*TisztaBal)* ∧
***amikor*** *OttBal* ∧ *TisztaJobb: K(TisztaJobb)* ∧
***amikor*** *OttBal* ∧ ¬*TisztaJobb: K(*¬*TisztaJobb)*

Aktív érzékelés esetén a tervkészítő a *Balra* cselekvés végrehajtása után nem tudja, hogy a bal oldali négyzet piszkos-e, ezért a cselekvésleírásban található két *TisztaBal* feltételes következményt elhagyhatjuk, hiszen a *TisztasagEllenorzes* tevékenység fogja visszaadni a négyzet

állapotát. Ezen cselekvések végrehajtása ugyanazt a két hiedelmi állapotot állítják elő, mint az automatikus érzékelés esetén elvégzett *Balra* művelet. A különbség abban mutatkozik meg, hogy aktív érzékelés használatakor a cselekvések egy hiedelmi állapotot egy hiedelmi állapotra transzformálják és csak az érzékelő cselekvések eredményeznek több hiedelmi állapotot.

Az előzőekben bemutatott feltételes tervkészítési módszer alapvető tesztfeladatokon jól működik, de a problémák többségéhez nem használható. Egy feltételes tervkészítő algoritmus összetettsége nagyobb, mint egy determinisztikus feladaton operáló algoritmusé és a problémák megoldásának előállítása is több időt vesz igénybe. Hatékonyabb megoldás lenne, ha a tervkészítési fázisban csak a fontosabb eseményekre készítenénk tervet és a többi eshetőséggel akkor foglalkoznánk, ha ténylegesen bekövetkeznek - ezt valósítja meg a következő módszer.

## 6.3. Végrehajtás monitorozás és újratervezés

Egy monitorozó ágens észlelni tudja, ha egy terv végrehajtása során eltérünk attól. Ez akkor következhet be, ha olyan rendkívüli események történnek, amelyekhez nincs megfelelő cselekvésleírásunk - ez a tervkészítési technika emiatt különösen fontos valós feladatok esetén. Kétféle monitorozási technikát fogunk majd megnézni:

- Cselekvésmonitorozás: a tervkészítő a környezetet ellenőrzi, hogy a következő cselekvés végrehajtható lesz-e.
- Tervmonitorozás: a tervkészítő a terv komplett hátralévő részét elemzi.

Ha probléma merül fel a végrehajtás alatt, akkor egy újratervező ágens fogja megmondani, hogy a váratlan helyzetek esetén milyen lépéseket kell tenni úgy, hogy meghívja a tervkészítőt egy módosított terv elkészítése végett.

A végrehajtás monitorozás és az újratervezés együttesen egy olyan módszer, ami részben és teljesen megfigyelhető világok esetén is használható szinte bármilyen reprezentáció mellett. Nézzünk meg egy szimpla algoritmust, amely cselekvésmonitorozást használ:

*UjratervezoAgens(erzekeles)* **returns** *cselekvés*
 *tb*: egy tudásbázis cselekvésleírásokkal; *terv* és *teljesterv*: kezdetben üres tervek;
 *cel*: egy cél

 Ertesit(*tb*, ErzekeloMondatKeszites(*erzekeles*, *t*))
 *aktualis* = LeirasKeszites(*tb*, *t*)
 **Ha** *terv* = [ ] **akkor**
  *teljesterv* = *terv* = Tervkeszito(*aktualis*, *cel*, *tb*)

**Ha** Elofeltetelek(Eloszor(*terv*)) nem igaz *tb*-ben **akkor**

*jeloltek* = Rendezes(*teljesterv*, az *aktualis*tól számított távolság szerint rendezve)

olyan *s* állapotot keres a *jeloltek*ből amelyekre

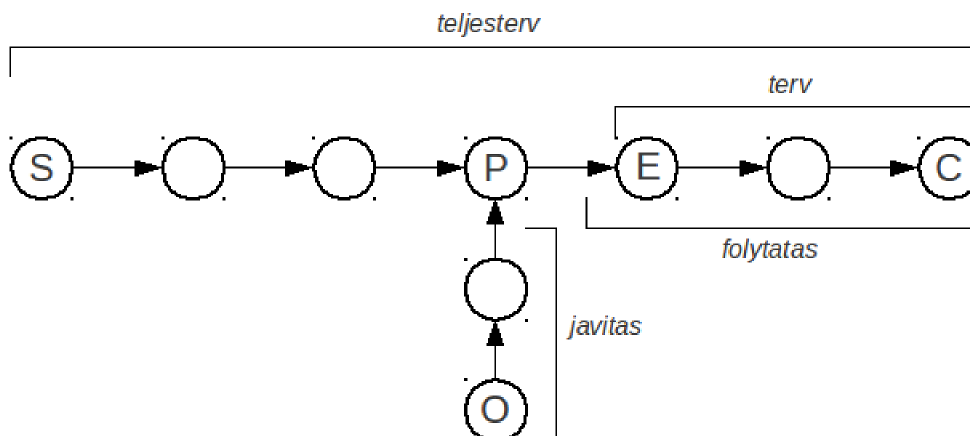
$kudarc \neq javitas$ = Tervkeszito(*aktualis*, *s*, *tb*)

*folytatas* = *teljesterv* *s*-től a végéig

*teljesterv* = *terv* = Hozzafuz(*javitas*, *folytatas*)

**return** RRT(*terv*)

A tervkészítő először egy *teljesterv*et készít a megadott célhoz, majd ezután az újratervező figyeli a *terv* eddig végrehajtott és hátralévő részét is a kiinduló *teljesterv*vel együtt. Az ágens minden cselekvés elvégzése előtt érzékeli, ha a terv további megvalósításának előfeltételei nem teljesülnek - ebben az esetben újratervezéssel olyan tervet állít elő, amely visszavezet a *teljesterv* egyik pontjára. Példaként vezessünk le egy ilyen folyamatot.




12. ábra: A cselekvésmonitorozás és újratervezés folyamata

A tervkészítő által elkészített teljes terv az *S* ponttól a *C* pontig vezető utat jelenti és tegyük fel, hogy jelenleg az *E* pontnál vagyunk. A következő lépésben az előfeltételek ellenőrzésénél viszont azt kapjuk, hogy valójában az *O* pontnál járunk. Ebben az esetben meg kell hívni a tervkészítő algoritmust, amely egy olyan *javitas* résztervet fog előállítani, amely visszavezet a *teljesterv* egy *s* állapotába - a példánkban ez az állapot a *P*. A módosított terv a *javitas* és a *folytatas* egyesítéséből jön létre és ezzel dolgozik tovább a tervkészítő.

A cselekvésmonitorozás hátránya, hogy nem tudja érzékelni, ha olyan állapotba jutunk ahonnan a terv fennmaradó része nem működik. A tervmonitorozás a teljes fennmaradó tervben lévő előfeltételeket megvizsgálja, ezzel egy hibás terv végrehajtását azonnal felfüggeszti.

További előnnyel jár, ha a tervmonitorozó algoritmust úgy implementáljuk, hogy az aktuális állapot helyett a célig vezető összes állapot esetén ellenőrizzük, hogy az aktuális állapot eleget tesz-e a terv előfeltételeinek - ekkor véletlenszerű sikert is elérhetünk. A porszívóvilág problémánk esetén például ez akkor fordulhat elő, ha mindkét négyzet piszkos és mielőtt az egyik négyzeten eltakarítanánk a piszkot, közben a másik négyzetet valaki megtisztítja.

Részlegesen megfigyelhető környezetek esetén számos probléma merül fel a módszerrel kapcsolatban. Lehetséges, hogy olyan hibák adódnak amiket a tervkészítő nem képest érzékelni, valamint az is előfordulhat, hogy az előfeltételek ellenőrzéséhez szükséges érzékelési cselekvések száma miatt a tervkészítő sosem jut el arra a pontra, hogy hasznos cselekvéseket hajtson végre. Ez utóbbi probléma az érzékelési cselekvések elvégzéséhez szükséges monitorozásból adódhat, amely újabb érzékelési cselekvéseket igényel. A tervkészítőnek ezért csak azokat a lényeges változókat kellene figyelnie, amelyek nagy eséllyel okoznak problémát a tervben és könnyen megfigyelhetőek.

Nem garantálható minden esetben az algoritmus sikere sem, mert nem korlátos nem determinisztikus környezet esetén az ágens zsákutcába juthat. Például a porszívóvilág esetén a tervkészítő nem tudhatja, hogy a porszívó elemei lemerülhetnek. Ha a zsákutcákat kizárjuk és feltételezzük, hogy egy terv végrehajtásakor mindig van lehetőség a cél eléréséhez, akkor az ágens minden esetben sikeres lesz. Egy újratervező és egy feltételes tervkészítő tudása éppen ezért megegyezik. Ha úgy módosítjuk a tervkészítőt, hogy csak egy részleges tervet állítson össze, akkor egy ilyen terv megoldása lehet az eredeti feladatnak is kisebb költség mellett. Végül még a helytelen cselekvés problémájával is számolnunk kell, ami azt jelenti, hogy a tervkészítő lépései sikertelenek egy ismeretlen előfeltétel vagy következmény miatt. Például, ha egy zárat egy nem megfelelő kulccsal próbálunk kinyitni, akkor a kinyitás részterv ismételgetése helyett egy másik kulcs megszerzése lenne triviális. Erre megoldást nyújhat egy olyan tervhalmaz, amiben javítótervek találhatóak, de bonyolultabb problémák esetén hatékonyabb módszer a tanulás. A tervkészítőnek ez utóbbi esetben tudnia kell módosítani a cselekvésleírásokat bizonyos számú próbálkozás után, hogy az újratervező ágens automatikusan új tervet tudjon készíteni - a példánkban ez egy másik kulcs megszerzésének a tervét jelenti.

# 7. Ütemezési feladatok

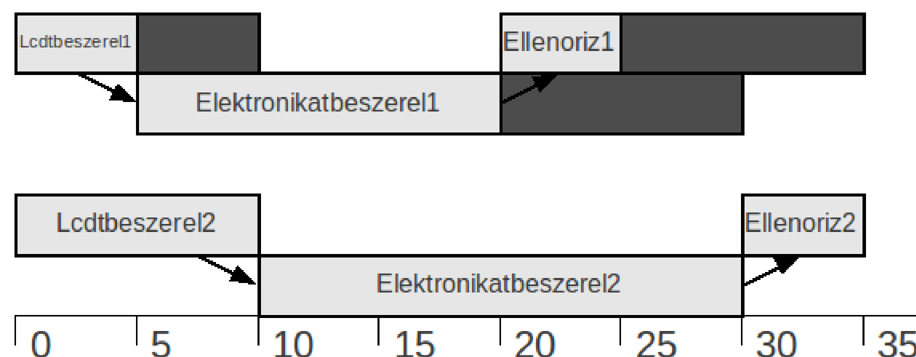
A STRIPS reprezentációval csak azt írjuk le, hogy mit csinálunk, de nem tartalmazza, hogy mikor kezdjük és milyen hosszú ideig tart egy cselekvés. Az ütemezési feladatok a lépések elvégzéséhez szükséges időre épülnek. Minden feladat elvégzésére adott mennyiségű idő és szabad erőforrás szükséges. A célunk, hogy a lehető legrövidebb idő alatt oldjuk meg a teljes problémát.

Az ütemezési feladatok leírása megegyezik a tervkészítési feladatokéval azzal a különbséggel, hogy jelölni kell az egyes cselekvések elvégzéséhez szükséges időtartamot. Ha adott a cselekvések egy részleges rendezése az időtartamokkal együtt, akkor alkalmazható a kritikus útvonal módszer. Ez lehetővé teszi a tevékenységek kezdési és befejezési időpontjainak meghatározását. Szemléltetésképpen deklaráljunk egy egyszerű ütemezési feladatot, amire a későbbiekben példaként is hivatkozhatunk:

*Kezdőállapot((Burkolat* $B_1$*)* $\wedge$ *(Burkolat* $B_2$*)* $\wedge$ *(Lcdpanel* $L_1$ $B_1$ *5)*
$\wedge$ *(Lcdpanel* $L_2$ $B_2$ *10)* $\wedge$ *(Elektronika* $E_1$ $B_1$ *15)* $\wedge$ *(Elektronika* $E_2$ $B_2$ *20))*
*Célállapot((Ellenőrizve* $B_1$*)* $\wedge$ *(Ellenőrizve* $B_2$*))*

*Cselekvés((Lcdtbeszerel l b),*
*Előfeltétel: (Lcdpanel l b i)* $\wedge$ *(Burkolat b)* $\wedge$ $\neg$*(Lcdbenne b)*
*Következmény: (Lcdbenne b)* $\wedge$ *(Elteltido i))*
*Cselekvés((Elektronikatbeszerel e b),*
*Előfeltétel: (Elektronika e b i)* $\wedge$ *(Burkolat b)* $\wedge$ *(Lcdbenne b)*
*Következmény: (Elektronikabenne b)* $\wedge$ *(Elteltido i))*
*Cselekvés((Ellenoriz b),*
*Előfeltétel: (Lcdbenne b)* $\wedge$ *(Elektronikabenne b)* $\wedge$ *(Burkolat b)*
*Következmény: (Ellenorizve b)* $\wedge$ *(Elteltido 5))*

A feladat kétféle televízió összeszereléséből áll. Újdonság az eddigi reprezentációkhoz képest a cselekvések időtartamának megadása - amit ebben az esetben percben adtuk meg.




13. ábra: Az összeszerelési feladat megoldása

Ha adott a cselekvések részleges rendezése, akkor a kritikus útvonal módszerrel egyszerűen kiszámíthatjuk ezek lehetséges kezdési és befejezési időpontjait. Az útvonal a cselekvések egy rendezett sorozata, a kritikus útvonal az az út, melyben, ha egy cselekvést késleltetünk, akkor

az egész terv több időbe fog kerülni. Jelen esetben a kritikus útvonal a kettes számú készülék összeszerelésének a terve. A minimális végrehajtási idő eléréséhez a kritikus útvonal cselekvéseit amint tudjuk azonnal el kell kezdenünk. Az ezen az útvonalon kívül eső cselekvéseknek továbbá létezhet egy időtartománya (a legkorábbi és a legkésőbbi kezdés idejét kell megadni - jelölésük: *ES* és *LS*), amin belül bármikor elkezdhető a végrehajtás, a teljes időtartam megváltozása nélkül. Az ábrán ezt a mozgásteret a sötét területek jelzik. Az *ES* és *LS* mérőszámok pontos definíciói a következők:

- *(ES indit)* = 0
- *(ES b)* = $max_{a \prec b}$*(ES a)* + *(Elteltido a)*
- *(LS befejez)* = *(ES befejez)*
- *(LS a)* = $min_{a \prec b}$*(LS a)* - *(Elteltido a)*

Az algoritmus első lépéseként az *(ES indit)* értéke legyen nulla. Ha elérünk egy olyan *b* cselekvést, hogy minden előtte lévő cselekvéshez már van hozzárendelve *ES* érték, akkor az *(ES b)* értéke a közvetlenül mögötte lévő cselekvések közül a legkorábbihoz tartozó befejezési érték maximuma lesz. A legkorábbi befejezési idő a cselekvés időpontjának és időtartamának az összege. Ezt addig folytatjuk, amíg minden cselekvésnek adtunk időpontot. Az *LS* értékeket ugyanezzel a módszerrel számoljuk ki az ellenkező irányból elindulva.

Az algoritmus komplexitása *cse*, ahol *cs* a cselekvések száma az *e* pedig a cselekvéshez tartózó maximális elágazások száma.

## 7.1. Ütemezés erőforráskorlátokkal

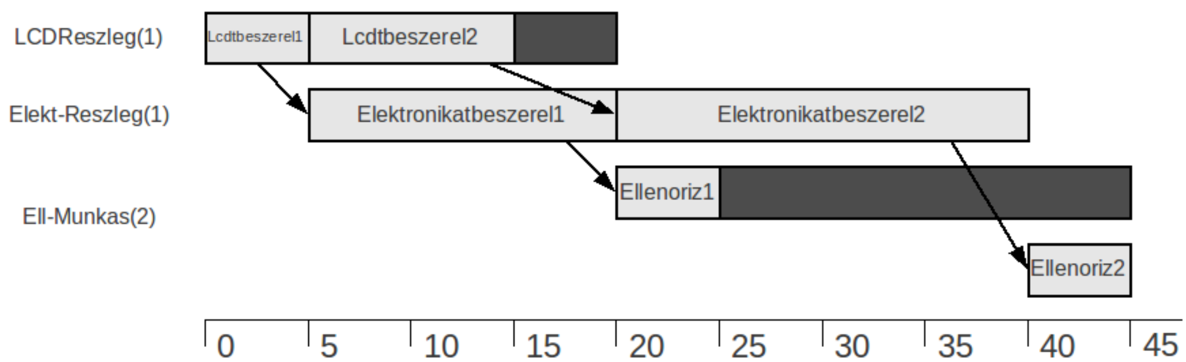
Valós problémák esetén ugyanakkor erőforráskorlátokkal is számolnunk kell. Általában újrahasznosítható erőforrásokkal találkozunk, ami azt jelenti, hogy csak a cselekvés ideje alatt foglalt az erőforrás és többször is használható. Új jelölés bevezetésére is szükségünk van: le kell írni, hogy milyen típusú és mekkora mennyiségű erőforrásra van szükség egy cselekvéshez. Az erőforrás előfeltétel és egyben következmény is, hiszen egy tevékenység megkezdéséhez szükséges valamilyen szabad erőforrás és a cselekvés alatt az erőforrás mennyisége átmenetileg csökken.

A feladatok komplexitásának csökkentése érdekében érdemes aggregációs technikát alkalmazni. Az objektumokat mennyiségekké csoportosíthatjuk abban az esetben, ha a feladat szempontjából nem megkülönböztethetőek. Például az ellenőrzést végző munkások közül teljesen mindegy, hogy ki vizsgálja át a televíziót, azaz az *(Ell-munkas* $E_1$*)* és *(Ell-munkas*$E_2$*)* elnevezések helyett az *(Ell-munkas 2)* jelölést használhatjuk. Az egyszerűsítés előnye abban mutatkozik

meg, hogy ha például 9 ellenőrzés cselekvésünk van, de csak 8 munkásunk akkor az algoritmus ezt azonnal felismeri és más megoldást keres. Az előző példánk kibővített leírása:

*Kezdőállapot((Burkolat* $B_1$*)* ∧ *(Burkolat* $B_2$*)* ∧ *(Lcdpanel* $L_1$ $B_1$ *5)*
∧ *(Lcdpanel* $L_2$ $B_2$ *10)* ∧ *(Elektronika* $E_1$ $B_1$ *15)* ∧ *(Elektronika* $E_2$ $B_2$ *20)*
∧ *(Lcdreszleg 1)* ∧ *(Elekt-reszleg 1)* ∧ *(Ell-munkas 2))*
*Célállapot((Ellenőrizve* $B_1$*)* ∧ *(Ellenőrizve* $B_2$*))*

*Cselekvés((Lcdtbeszerel l b),*
*Előfeltétel: (Lcdpanel l b i)* ∧ *(Burkolat b)* ∧ ¬*(Lcdbenne b)*
*Következmény: (Lcdbenne b)* ∧ *(Elteltido i)*
*Erőforrás: (Lcdreszleg 1))*
*Cselekvés((Elektronikatbeszerel e b),*
*Előfeltétel: (Elektronika e b i)* ∧ *(Burkolat b)* ∧ *(Lcdbenne b)*
*Következmény: (Elektronikabenne b)* ∧ *(Elteltido i)*
*Erőforrás: (Elekt-reszleg 1))*
*Cselekvés((Ellenoriz b),*
*Előfeltétel: (Lcdbenne b)* ∧ *(Elektronikabenne b)* ∧ *(Burkolat b)*
*Következmény: (Ellenorizve b)* ∧ *(Elteltido 5)*
*Erőforrás: (Ell-munkas 1))*




14. ábra: Az erőforráskorlátokat is tartalmazó feladat megoldása

Az erőforráskorlátok bevezetése bonyolulttá teszi az ütemezési feladatokat, különösen a legrövidebb befejezési időt adó ütemezés megtalálása nehéz, ezért elengedhetetlen egy jó heurisztika a megoldáskeresésben. Egy egyszerű, de nem tökéletes heurisztika a minimális tartalék algoritmus. Az eljárás minden lépésben azt a cselekvést ütemezi, amelynek az összes előzmé-

nyét már ütemeztük és a legkevesebb tartalék ideje van a legkorábbi kezdéshez képest. Az *ES* és *LS* értékek minden iterációban újra számításra kerülnek.

Az itt bemutatott módszer két részre bontja a probléma megoldását: először egy részben rendezett tervet készítünk, majd ezután ezt módosítjuk a határidő- és erőforráskorlátokat figyelembe véve. Egy sok erőforrás megkötést tartalmazó feladat esetén bizonyos tervek sokkal hatékonyabb ütemezést biztosítanak, ezért a két fázist célszerű integrálni, hogy a cselekvések időkorlátait már az első fázisban figyelembe lehessen venni. A részben rendezett tervkészítő algoritmusok kibővíthetőek, hogy az okozati kapcsolatok veszélyhelyzeteinek észleléséhez hasonlóan az erőforráskorlátok konfliktusait is felismerjék, valamint a heurisztikák a terv végrehajtási idejét is megjósolhassák.

# 8. Alkalmazások

A cselekvéstervezés fő felhasználási területei a repülőgépipar, az űrkutatás és a katonai logisztika. Az ezekhez a területekhez kötődő feladatok azonban nem pusztán tervezési problémák, egyesítik a hosszú távú tervezést a rövid távú ütemezéssel és a tervek folyamatos ellenőrzésen esnek át, hiba észlelése esetén pedig újra kell tervezni.

A NASA többek között az űrsiklók karbantartási munkálataihoz, az autonóm űrszondák irányítására, illetve a Hubble űrtávcső működéséhez használ tervezőrendszereket, melyek saját fejlesztésűek. Az űrszondák irányításánál kritikus feladat a tervek monitorozása a gyakori újratervezések szükségessége miatt. A Hubble esetén sok beérkező igényt kell összehangolni hatékony módon, biztosítani kell az erőforrások optimális kihasználását és a váratlan eseményeket is kezelni kell. Az Európai Űrügynökség saját tervező és ütemező rendszere segítségével támogatja a fejlesztési feladatait és a repülőgépek összeszerelését. Az Amerikai Egyesült Államok hadserege az iraki háború idején hadműveletek és evakuálások tervezéséhez sikeresen alkalmazta a SIPE-2 elnevezésű rendszerét.

Klasszikus tervezési módszereket használnak a képfeldolgozásban, ahol különböző programelemek együttműködését kell biztosítani előzetes elemzések eredményei alapján. Hollandiában ütemezési feladattal készítik el a vasutak személyzetének napirendjét.

Mindezek ellenére kevés helyen alkalmaznak tervkészítést az üzleti életben, viszont az ütemező rendszereket sikeresen alkalmazzák számos területen. Ennek egyik oka, hogy a tervezés értelemszerűen nehezebb feladat, hiszen ütemezés esetén nem kell eldönteni, hogy mit kell tenni, másrészt ezen döntések meghozatala a vállalatirányítók hatásköre.

## Saját alkalmazások

### Shakey világa PDDL-ben

```
(define (domain shakey)
  (:predicates (SZOBA ?x) (LAMPA ?x) (DOBOZ ?x) (ROBOT ?x)
    (ott ?x ?y) (rajta ?x ?y) (padlon ?x) (vilagit ?x))

  (:action megy
    :parameters ( ?mi ?honnan ?hova)
    :precondition (and (ott ?mi ?honnan) (ROBOT ?mi) (SZOBA ?hova) (padlon ?mi))
    :effect (and (ott ?mi ?hova) (not (ott ?mi ?honnan))))

  (:action tol
    :parameters ( ?mi ?mit ?honnan ?hova
    :precondition (and (ott ?mit ?honnan) (ott ?mi ?honnan) (DOBOZ ?mit)
                  (ROBOT ?mi) (SZOBA ?hova) (padlon ?mi))
    :effect (and (ott ?mit ?hova) (ott ?mi ?hova) (not (ott ?mit ?honnan))
                  (not (ott ?mi ?honnan))))

  (:action felmaszik
    :parameters ( ?mi ?hova ?honnan)
    :precondition (and (ott ?mi ?honnan) (ott ?hova ?honnan) (DOBOZ ?hova)
                  (ROBOT ?mi))
    :effect (and (rajta ?mi ?hova) (not (padlon ?mi))))

  (:action lemaszik
    :parameters ( ?mi ?honnan)
    :precondition (and (rajta ?mi ?honnan) (DOBOZ ?honnan) (ROBOT ?mi))
    :effect (and (not (rajta ?mi ?honnan)) (padlon ?mi)))

  (:action felkapcsol
    :parameters ( ?mi ?mit ?mirol ?honnan)
    :precondition (and (ott ?mit ?honnan) (rajta ?mi ?mirol) (DOBOZ ?mirol)
                  (LAMPA ?mit) (ROBOT ?mi))
    :effect (and (vilagit ?mit)))
```

```
(:action lekapcsol
    :parameters ( ?mi ?mit ?mirol ?honnan)
    :precondition (and (ott ?mit ?honnan) (rajta ?mi ?mirol) (DOBOZ ?mirol)
                  (LAMPA ?mit) (ROBOT ?mi))
    :effect (and (not(vilagit ?mit)))

(define (problem shakey)
   (:domain shakey)
   (:objects szoba1 szoba2 szoba3
      doboz1 doboz2 doboz3
      lampa1 lampa2 lampa3
      shakey)

   (:init (DOBOZ doboz1) (DOBOZ doboz2) (DOBOZ doboz3)
      (LAMPA lampa1) (LAMPA lampa2) (LAMPA lampa3)
      (SZOBA szoba1) (SZOBA szoba2) (SZOBA szoba3)
      (ROBOT shakey) (padlon shakey)
      (ott doboz1 szoba1) (ott doboz2 szoba1) (ott doboz3 szoba1)
      (ott lampa1 szoba1) (ott lampa2 szoba2) (ott lampa3 szoba3)
      (ott shakey szoba2)

   (:goal (and (ott doboz1 szoba2) (ott shakey szoba1)
      (vilagit lampa2))))
```

A lokális keresést használó LPG tervkészítő program a következő tervet állítja elő:

*0: (MEGY SHAKEY SZOBA2 SZOBA1)*
*1: (TOL SHAKEY DOBOZ1 SZOBA1 SZOBA2)*
*2: (FELMASZIK SHAKEY SZOBA2 DOBOZ1)*
*3: (FELKAPCSOL SHAKEY LAMPA2 DOBOZ1 SZOBA2)*
*4: (LEMASZIK SHAKEY DOBOZ1)*
*5: (MEGY SHAKEY SZOBA2 SZOBA1)*

**Shakey világa PROLOG nyelven**

```
keszit(Terv):-
    write('Kezdőállapot:'),nl,
    Kezdo = [ott('shakey', 'szoba3'), ott('doboz1','szoba1'), ott('doboz2','szoba1'),
    ott('doboz3','szoba1'), doboz('doboz1'), doboz('doboz2'), doboz('doboz3'),
    lampa('lampa1'), ott('lampa1', 'szoba1'), lampa('lampa2'), ott('lampa2', 'szoba2'),
    lampa('lampa3'), ott('lampa3', 'szoba3'), robot('shakey'), rajta('shakey', 'padló')],
    write(Kezdo), nl, nl, write('Célállapot:'),nl,
    Cel = [ott('shakey', 'szoba2'), ott('doboz1','szoba2'), ott('doboz2','szoba1'),
    ott('doboz3','szoba1'), doboz('doboz1'), doboz('doboz2'), doboz('doboz3'),
    lampa('lampa1'), ott('lampa1', 'szoba1'), lampa('lampa2'), ott('lampa2', 'szoba2'),
    lampa('lampa3'), ott('lampa3', 'szoba3'), vilagit('lampa2'), robot('shakey'),
    rajta('shakey', 'padló')],
    write(Cel),nl,
    megold(Kezdo,Cel, [], Terv).

operator(megy(Mi, Honnan, Hova), % név
    [ott(Mi, Honnan), rajta(Mi, 'padló'), robot(Mi)], % előfeltételek
    [ott(Mi, Hova)], % állapot hozzáadása
    [ott(Mi, Honnan)]). % állapot törlése

operator(tol(Mit, Honnan, Hova),
    [ott(Mit, Honnan), ott('shakey',Honnan), rajta('shakey', 'padló'), doboz(Mit)],
    [ott(Mit, Hova), ott('shakey', Hova)],
    [ott(Mit, Honnan), ott('shakey', Honnan)]).

operator(felmaszik(Mire),
    [ott('shakey', Hely), rajta('shakey', 'padló'), ott(Mire, Hely), doboz(Mire)],
    [rajta('shakey', Mire)],
    [rajta('shakey', 'padló')]).

operator(lemaszik(Mirol),
    [rajta('shakey', Mirol), doboz(Mirol)],
    [rajta('shakey', 'padló')],
    [rajta('shakey', Mirol)]).
```

```
operator(felkapcsol(Mit, Honnan, Mirol),
    [rajta('shakey', Mirol), ott(Mit, Honnan), ott('shakey', Honnan), doboz(Mirol),
     lampa(Mit)],
    [vilagit(Mit)], []).

operator(lekapcsol(Mit, Honnan, Mirol),
    [rajta('shakey', Mirol), ott(Mit, Honnan), ott('shakey', Honnan), doboz(Mirol),
     lampa(Mit)], [],
    [vilagit(Mit)]).

reszhalmaz([H|T], Halmaz):-
    member(H, Halmaz),
    reszhalmaz(T, Halmaz).
reszhalmaz([], _).

torol_lista([H|T], Jelenlegiallapot, Ujallapot):-
    torol(H, Jelenlegiallapot, Maradek),
    torol_lista(T, Maradek, Ujallapot).
torol_lista([], Jelenlegiallapot, Jelenlegiallapot).

torol(X, [X|T], T).
torol(X, [H|T], [H|R]):-
    torol(X, T, R).

megold(Allapot, Cel, Terv, Terv):-
    reszhalmaz(Cel, Allapot),nl, % célfeltételek ellenőrzése
    kiir(Terv).

megold(Allapot, Cel, Eddig, Terv):-
    operator(Nev, Elofeltetelek, Hozzaad, Torol), % első operátor kiválasztása
    \+member(Nev, Eddig), % kör ellenörzése
    reszhalmaz(Elofeltetelek, Allapot), % előfeltételek ellenőrzése
    torol_lista(Torol, Allapot, Maradek), % régi állapotok törlése
    append(Hozzaad, Maradek, Ujallapot), % új állapotok hozzáadása
    megold(Ujallapot, Cel, [Nev|Eddig], Terv). % rekurzió
```

```
kiir([]).
kiir([H|T]):-
    kiir(T),
    write(H), nl.
```

A fenti PROLOG nyelvben írt programom egy tervet állít elő egy robot számára a 2.3-as fejezetből már ismert Shakey problémához, azzal a különbséggel, hogy Shakeynek nem kell az egyes számú szobába mennie. A program egy egyszerű progresszív tervkészítő algoritmussal dolgozik, amely halmazműveleteket használ; nézzük meg, hogy hogyan működik. A kezdő- és a célállapot predikátumok és állapotok felsorolásával van megadva. A cselekvések neve és előfeltételei után külön halmazban vannak felsorolva azok a hatások, amelyeket hozzá kell adni és amelyeket törölni kell az aktuális állapotból. Első lépésként ellenőrizni kell, hogy az aktuális állapot a célállapot-e. Ha igen, akkor az eddig alkalmazott operátorok listája lesz a terv, ellenkező esetben pedig egy cselekvést kell választani. A tevékenység kiválasztása után ellenőrizni kell, hogy alkalmaztuk-e már a cselekvést és hogy a szükséges előfeltételek fent állnak-e az adott állapotban. Ha ezek közül valamelyik nem teljesül, akkor másik cselekvést kell választani, különben pedig az operátor leírásának megfelelően módosítani kell a világ állapotát és a már használt operátorok listájához hozzá kell adni az operátort. Az eljárás ezután tovább folytatódik az első lépéstől addig, amíg el nem érjük a célállapotot. A feladatunk esetén a következő operátor sorozat lesz az eredmény, amelyet visszafele olvasva kapjuk meg a tervet:

*Terv = [lemaszik(doboz1), felkapcsol(lampa2, szoba2, doboz1),*
*felmaszik(doboz1), tol(doboz1, szoba1, szoba2), megy(shakey, szoba3, szoba1)]*

**Értékelés**

Láthatjuk, hogy a legalapvetőbb algoritmusok is alkalmasak tervkészítésre és gyorsan megtalálják a megoldást, igaz, csak nagyon egyszerű problémák esetén. Komplexebb feladatoknál a heurisztika hiánya miatt könnyen kudarcba fulladhat a keresés az állapottér nagysága miatt, hiszen az sem lenne biztos, hogy véges időn belül megtalálnánk a megoldást. Valójában a bonyolultabb eljárások is alapvetően hasonló módon működnek, de számos, a hatékonyságot növelő módszert alkalmaznak. Mivel a tervkészítésben kritikus tényező, hogy egy megoldást a lehető legrövidebb idő alatt állítsunk elő, ezért komplex problémák esetén elengedhetetlen egy jó heurisztika használata az algoritmusban.

# 9. Összefoglalás

Először áttekintettük a tervezési modelleket. Egy tervezési feladatot szemlélhetünk keresési problémaként és logikai következtetésként is, de a ma használt alapfogalmak és feltevések a klasszikus modellből erednek. Ez a modell egy teljesen kiszámítható, diszkrét világot feltételez diszkrét állapotokkal, amiben egy probléma megoldását a kezdőállapotból a végállapotba juttató cselekvéssorozat jelenti. A cselekvéseket operátorokkal írjuk le, amelyek tartalmazzák a tevékenység előfeltételeit és következményeit is.

Ezután megnéztük, hogy milyen módszerekkel reprezentálhatjuk a problémákat. A két legelterjedtebb nyelv a tervkészítési feladatok leírására a STRIPS és a PDDL. A STRIPS reprezentáció állapotok, célok és cselekvések leírásából áll és azt feltételezi, hogy egy cselekvés végrehajtásakor csak azok az állapotok változnak meg, amelyeket a tevékenység következményei deklarálnak. A PDDL a leíró nyelvek egységesítése miatt született meg, tartalmazza a STRIPS összes kiterjesztését és a problémaleírásokat két részre bontja: egy tárgyterületi és egy feladat részre. Utóbbi tartalmazza az objektumok és a kezdő- és célállapot definícióit.

A következő kisebb részben a tervező algoritmusokról ejtünk pár szót általánosságban. Az eljárások három bemenetet várnak el: a probléma környezetének, a cselekvő céljának, valamint az elvégezhető cselekvések leírását. A tervezési problémák egyszerűsítése miatt korlátozó feltételezéseket is használnunk kell.

A továbbiakban a tervezési algoritmusokat elemeztük részletesen. Megnéztünk két állapottér keresőt: a progresszív és regresszív keresési módszert. Az első algoritmus előrefelé keres a megadott kezdőállapottól addig, amíg olyan állapotot nem talál, hogy az kielégíti a célfeltételeket. A regresszív módszer a céltól halad a kezdeti állapot felé, tehát olyan cselekvéseket választ ki, amelyek hatása megegyezik a célok halmazának egyik elemével. A regressziós tervkészítés elágazási tényezője általában kisebb, ezért a futási ideje gyorsabb a progresszív keresőénél.

Ezután megismerkedtünk egy másfajta szemléletmóddal, a tervtérben való kereséssel, ahol a gráf csomópontjai részlegesen definiált terveket, az élek pedig tervfinomító műveleteket jelentenek és a célállapot maga a megoldás. A tervek emellett nem csak teljesen rendezettek, hanem részben rendezettek is lehetnek, ami azt jelenti, hogy csak az elengedhetetlen döntéseket hozzuk meg a tervkészítés alatt - ez a legkisebb elkötelezettség tervezési technika. Egy részben rendezett terv ábrázolása egy $\langle CS, M, K \rangle$ elemhármassal történik, ahol a $CS$ a cselekvések halmazát jelöli, az $M$ a rendezési megszorításokat tartalmazza, a $K$ pedig okozati kapcsolatok együttese. Ezzel a reprezentációval tud dolgozni a POP algoritmus, amely egy regresszív részben rendezett tervező. Az algoritmus a kisebb elágazási tényező miatt hatékonyabb az állapottér-keresőknél.

Nem determinisztikus környezetek esetén a tervezőnek követnie kell az eseményeket a terv megvalósítása közben és váratlan helyzetek esetén módosítania kell a tervet, valamint figyelem-

be kell vennie a hiányos és hibás információkat is. A hiányzó ismeretek feltárásának lehetőségét a világ nemdeterminisztikussága határozza meg. Korlátos nemdeterminisztikusság esetén a cselekvések kimenetele bizonytalan, de a következményeket megadhatjuk a cselekvés leírásában. Nem korlátos nemdeterminisztikusság esetén az előfeltételek és következmények halmaza ismeretlen vagy túl nagy ahhoz, hogy felsoroljuk őket. A tervkészítési módszerek közül kettőt néztünk meg: a feltételes tervkészítést, amely korlátos nemdeterminisztikusság esetén használható és a végrehajtás monitorozást és újratervezést, ami nem korlátos nemdeterminisztikusság esetén alkalmazható. A feltételes tervkészítés alapvető jellemzője, hogy a terv bizonyos pontjain ellenőrzi a világban bekövetkező eseményeket, ezzel biztosítva a bizonytalanságkezelést. Először teljesen, majd részlegesen megfigyelhető környezetekben néztük meg a módszert, valós problémák esetén az utóbbi fordul elő gyakrabban. A feltételes tervkészítés alapvető tesztfeladatokon jól működik, de a problémák többségéhez nem használható. Egy feltételes tervkészítő algoritmus összetettsége nagyobb, mint egy determinisztikus feladaton működő algoritmusé és a problémák megoldásának előállítása is több időt vesz igénybe. A végrehajtás monitorozás és az újratervezés együttesen egy olyan módszer, ami részben és teljesen megfigyelhető világok esetén is használható szinte bármilyen reprezentáció mellett. A kiterjesztett algoritmusnak is vannak hátrányai: nem hatékony valós környezetekben és képtelen saját célokat kialakítani. Erre jelent megoldást a folytonos tervkészítés, amely a tervkészítést és a cselekvésmonitorozást egy folyamatban egyesíti, bármeddig kitart egy környezetben és nagyban hasonlít a részben rendezett tervkészítéshez - ez a tervkészítési technika már valós problémák esetén is használható.

A következő részben szó esett az ütemezési feladatokról, amelyekben idő és erőforráskorlátokkal is számolnunk kell. Az ilyen problémákhoz először egy részben rendezett tervet készítünk és ezután ezt módosítjuk a idő- és erőforráskorlátokat figyelembe véve. A részben rendezett tervkészítő algoritmusok kibővíthetőek, hogy az erőforráskorlátok konfliktusait felismerjék és megjósolhassák a terv végrehajtási idejét is. Végül az utolsó fejezetben láttunk néhány példát a gyakorlati alkalmazásokra és rámutattunk, hogy az ezen a területeken használt tervkészítő rendszerekben nélkülözhetetlen egy hatékony heurisztika alkalmazása.

# 10. Irodalomjegyzék

[1] Dan Weld : An Introduction to Least-Commitment Planning, AI Magazine, 1994.
[2] Mesterséges Intelligencia (szerk. Futó Iván), Aula Kiadó, 1999.
[3] Stuart J. Russel – Peter Norvig : Mesterséges Intelligencia modern megközelítésben, Panem Könyvkiadó, 2005.
[4] Dr. Kovács Szilveszter : Intelligens rendszerek
http ://users.iit.uni-miskolc.hu/ szkovacs/GAMFIR/IRE2.pdf
[5] Malte Helmert : An Introduction to PDDL, http ://www.cs.ust.hk/ qyang/221/introtopddl2.pdf
[6] A. Newell - J.C. Shaw - Herbert Simon : Report on a general problem-solving program, 1959.
[7] SWI-Prolog, http ://www.swi-prolog.org/
[8] LPG-td : A fully-automated domain-independent planner for PDDL2.2 domains
http ://zeus.ing.unibs.it/lpg/