Plánování

- Máme stavy, operátory, počátek, cíl.
- Otázka zní, jak použít operátory, abychom dosáhli stavu, který je cílový.

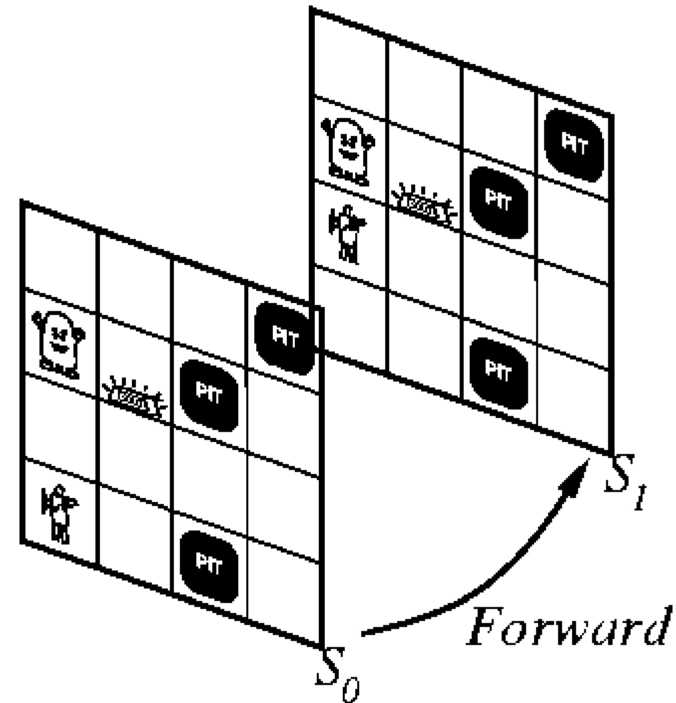
Už známe:

- Prohledávání stavového prostoru.
- Situační kalkul - popíši svět v logice a použiji rezoluční metody, ať za mě prohledává.

Situační kalkul

- Fakta platí v situacích např. $holding(gold, now)$
- Situace jsou propojeny predikátem $result(a, s)$, který definuje výsledek akce a v situaci s.

Šikovný, ale potřebujeme příliš mnoho axiomů pro reprezentaci rámce, proto se podíváme na specializované metody



STRIPS, Shakey

Vlastnosti	Akce
At(Shakey,x), In(x,r) & In(y,r) & On(Shakey, Floor)	Go(y)
Pushable(b),At(Shakey,x), In(x,r) & In(y,r)	Push(b,x,y)
Climbable(b), At(Shakey,x) &At(b,x)	Climb(b)
	Down(b)
	TurnOn(ls)
	TurnOff(ls)

STRIPS ve světě kostek

Akce:

- UNSTACK(A,B)
- STACK(A,B)
- PICKUP(A) - ze stolu
- PUTDOWN(A) - na stůl

Popis prostředí:

- ON(A,B)
- ONTABLE(A)
- CLEAR(A)
- HOLDING(A)
- ARMEMPTY

STRIPS - pokrač.

Axiomy:

$$(\exists x : HOLDING(x)) \longleftrightarrow \neg ARMEMPTY$$

$$\forall x : (ONTABLE(x) \longleftrightarrow \neg \exists y : ON(x, y))$$

$$\forall x : (\neg \exists y : ON(y, x)) \longleftrightarrow CLEAR(x))$$

Plný popis akcí

akce	PRECONDITIONS	DELETE	ADD
UNSTACK(x,y)	ON(x,y) & CLEAR(x) & ARMEMPTY	ON(x,y) & ARMEMPTY	HOLDING(x) & CLEAR(y)
STACK(x,y)	CLEAR(y) & HOLDING(x)	CLEAR(y) & HOLDING(x)	ARMEMPTY & ON(x,y)
PICKUP(x)	CLEAR(x) & ONTABLE(x) & ARMEMPTY	ONTABLE(x) & ARMEMPTY	HOLDING(x)
PUTDOWN(x)	HOLDING(x)	HOLDING(x)	ONTABLE(x) & ARMEMPTY

Plánovač

Plánovač obsahuje:

- **zásobník cílů**, na kterém jsou cíle i operátory navržené pro splnění těchto cílů
- databázi popisující současnou situaci
- množinu operátorů
- (axiomy popisující prostředí pro odvození vlastností, které nejsou řečeny explicitně)

Pozn: jedná se vlastně o zpětné prohledávání do hloubky, kde máme zjednodušený výpočet rámce následujícího stavu ve situačním kalkulu.

Algoritmus **STRIPS**

```
push(GOAL,STACK)
while not empty(STACK)
G←pop(STACK)
if G = akce then PERFORM(C)
elseif G = conjunction then
    if G is not true in the current database
        push(G,STACK)
        foreach L literal of G that is not true in the current database
            push(L,STACK)
    endif
else /* G is a subgoal*/
    if G not true in current database then
        choose an action S with G as its effect
        push(G,STACK)
```

```
        push(PRECONDITIONS(G),STACK)
    /* testuj slepé cesty a backtrack*/
    endif
endwhile
```

Složky plánovacího systému

1. vybrat nejvhodnější pravidlo (analýza prostředků a cílů)
2. aplikovat vybrané pravidlo a vypočíst nový stav úlohy
3. zjistit, zda bylo nalezeno řešení
4. najít slepé cesty
5. detekovat skoro správné řešení (a aplikovat opravné techniky)

Nelineární plánování

Plánování jakožto prohledávání prostoru plánů

POP (partially ordered plan)

Částečně uspořádaný plán sestává z:

- Kroků plánu (STEPS). Každý krok je jedna instance operátoru.
- Podmínky na uspořádání (ORDERINGS). Každá podmínka říká, že jeden krok musí předcházet jinému kroku, zapisujeme $S_i \prec S_j$.
- Dosazení za jednotlivé proměnné. Každé přiřazení je typu $v = x$, kde v je proměnná v některém kroku a x je buď konstanta, nebo jiná proměnná.
- Množinu kauzálních hran $S_i \overrightarrow{c} S_j$ s významem S_i splní c pro S_j. Kauzální harny slouží k pamatování důvodu kroků v plánu – důvod přítomnosti S_i v plánu je splnění předpokladu c pro S_j.[a]

[a] Občas se nazývají protection intervals.

Obrázek: minimální plán, ponožky a boty, linearizace plánu

Řešením úlohy plánování je najít takový plán, který garantuje splnění cíle. Plán necháváme v podobně neúplně uspořádané a instancializované, aby měl prováděcí agent více volnosti, např. pro paralelismus či kombinaci s dalšími plány.

Řešením je **úplný a konzistentní plán**. Úplný plán je takový, že každý předpoklad každého kroku je splněne nějakým dalším krokem. Krok splňuje podmínku pokud je podmínka jedním z jeho efektů a žádný další krok nemůže zrušit platnost této podmínky.

Formálně, krok S_i plní podmínu c kroku S_j pokud

- $S_i \prec S_j$ a $c \in EFFECTS(S_i)$
- neexistuje krok S_k takový, že $\neg c \in EFFECTS(S_k)$ a $S_i \prec S_k \prec S_j$ je konzistentní linearizace plánu.

Konzistentní plán je plán bez sporu v uspořádání a přiřazení proměnných.

Příklad: koupit mléko, banány a vrtačku, operátory: Go pro přesun, Buy pro koupi. HWS je HW obchod, SM je supermarket, At je predikát na pamatování si pozice, Sells reprezentuje znalost, kde se co prodává.

Theorem 1 (Korektnost a úplnost POP) *Algoritmus POP je korektní a úplný, pokud používáme prohledávání do šířky či iterativně prohlubující se prohledávání v bodech volby* ***choose****.*

Algoritmus **POP (základní)**

function POP(initial, goals, *operators*) **returns** plan
plan=MAKE–MINIMAL–PLAN(initial, goals)
loop do
Termination: **if** there are no insatisfied preconditions **then return** plan
Subgoal selection: find a plan step S_{need} with an open precondition c
Action selection:
choose a step S_{add} from operators or STEPS(plan) that has c as an effect
if there is no such step **then fail**

add the causal link $S_{add} \xrightarrow{c} S_{need}$ to LINKS(plan)

add the ordering constraint $S_{add} \prec S_{need}$ to ORDERINGS(plan)
if S_{add} is a newly added step from *operators* **then**
add S_{add} to STEPS(plan)

add $Start \prec S_{add} \prec Finish$ to ORDERINGS(plan)

Therat resolution:

for each S_{threat} that threatens a link $S_i \xrightarrow{c} S_j$ in LINKS(plan) **do**

choose either

Promotion: Add $S_{threat} \prec S_i$ to ORDERINGS(plan)

*Demotion:*Add $S_{threat} \succ S_j$ to ORDERINGS(plan)

if not CONSISTENT(plan) **then fail**

endloop

Modifikace pro proměnné

Algoritmus **CHOOSE–OPERATOR s unifikací**

procedure CHOOSE–OPERATOR(plan, *operators*, S_{need},c)

 choose a step S_{add} from operators or STEPS(plan) that has c_{add} as an effect

  such that $u = UNIFY(c, c_{add}, BINDINGS(plan))$

 if there is no such step **then fail**

 add u to $BINDINGS(plan))$

 add $S_{add} \xrightarrow{c} S_{need}$ to LINKS(plan)

⋮

Algoritmus **RESOLVE THREATS –resolve later verze**

$\vdots$

$c^{|}$ is an effect of S_{threat} that threatens c in the link $S_i \overset{c}{\longrightarrow} S_j$

if $SUBST(BINDINGS(plan), c) = SUBST(BINDINGS(plan), \neg c^{|})$ **then**

choose either

Promotion: Add $S_{threat} \prec S_i$ to ORDERINGS(plan)

*Demotion:*Add $S_{threat} \succ S_j$ to ORDERINGS(plan)

if not CONSISTENT(plan) **then fail**

$\vdots$

V praxi použité plánovače

O–Plan (Currie and Tate, 1991) – otevřená plánovací architektura podobná POP. Na ní založen OPTIMUM–AVI a plánování v Hitachi továrně.

OPTIMUM–AVI byl používaný European Space Agency na podporu montáže, integrace a verifikace vesmírných lodí. Byl používán pro tvorbu plánu i monitoring. *Schopnost rychlého přeplánování se ukázala být klíčová.*

Společně s plánováním se často používají nástroje na job–sheduling, jako např. PERT grafy či critical path method. Tyto plány ale neobsahují kauzální vazby mezi akcemi a proto nejsou schopny změnit plán a lidské přeplánování je často velmi pomalé.

Hubble space telescope

Hubblův teleskop je dobrý příklad nutnosti automatického plánování.

Každý astronom může navrhnout pozorování. Ta se pak označí jako high priority (ty zaberou cca. 70% času), low priority (možná se provedou, možná ne), a zamítnutá. Každý návrh obsahuje strojově čitelnou specifikaci, co má být nasměrováno na které místo nebo a jaký typ expozice má být proveden. Některá pozorování mohou být provedena vždy, některá jen, je–li družice v zemském stínu, některá periodicky s tím, že všechna mají být ve stejném zastínění.

SIPE

SIPE – pracoval se stavy, ne logickým popisem. Pro návrh stavby mnohapatrové budovy. Pracoval příliš dlouho, $O(n^{2.5})$ kde n je počet pater. Rozumně by měl být mnohem blíže $O(n)$, protože skoro vše stačí plánovat pro jednotlivá patra samostatně.

Pro reálné aplikace plánovač potřebuje:

1. hierarchické plány
2. komplexnější podmínky, univerzální kvantifikaci ”Start vynese VŠECHNY věci na orbitu”. Podobně je třeba podmíněné operátory ”jde–li vše dobře, start vynese vše na orbitu, jinak to hodí do moře”.
3. čas – věci musí být hotové v daný čas, akce mají dobu trvání, některé stroje jsou dosažitelné jen v určitou dobu atd.
4. zdroje = finance, počet zaměstnanců, počet dílen a testovacích stanic.

Hierarchická dekompozice

Ve složitějších problémech plánování se neobejdeme bez hierarchické dekompozice. Ta nám umožní převádět plány na vyšší hladině abstrakce na plány konkrétnější. Abstraktní plán je např.
$[Go(Supermarket), Buy(Milk), Buy(Bananas), Go(Home)]$,
plán na nejnižší úrovni může vypadat:
$[Forward(1cm), Turn(1deg), Forward(1cm), \ldots]$
Přímým hledáním plánu na nejnižší úrovni abstrakce bychom pravděpodobně nedospěli k řešení, protože bychom museli prohledávat příliš mnoho možností – tento plán může obsahovat tisíce kroků.

Při hierarchické dekompozici zavedeme abstraktní operátory a pro každý abstraktní operátor jednu či více dekompozic, tj. skupin kroků které implementují plán pro daný operátor. Kromě abstraktních operátorů máme operátory primitivní.

Plán je úplný pokud je každý jeho krok primitivní operátor, takže může být přímo proveden agentem.

Dekompozice má větší efekt, pokud je pro abstraktní akce více možných dekompozicí – víc prohledávání se udělá na vyšší úrovni. Například ”Postav zdi” lze provést ze dřeva, cihel, betonu či nějakého sandwitche.

Př. špatné dekompozice – devítka.

My nyní:

- rozšíříme jazyk o abstraktní operátory a jejich dekompozice
- rozšíříme POP algoritmus o dekompozici abstraktních operátorů

Operátor Decompose(o,p) rozloží operátor o na plán p, například:

Decompose(Construction,
    Plan(STEPS:$\{S_1$:Build(Foundation),S_2:Build(Frame),
        S_3:Build(Roof), S_4:Build(Walls),
        S_5:Build(Interior)$\}$
    ORDERINGS:$\{S_1 S_2 \prec S_3 \prec S_5$,
        $S_2 \prec S_4 \prec S_5\}$
    BINDINBS:$\{\}$
    LINKS:
$\{S_1 \overrightarrow{Foundations} S_2, S_2 \overrightarrow{Frame} S_3, S_2 \overrightarrow{Frame} S_4, S_3 \overrightarrow{Roof} S_5, S_4 \overrightarrow{Walls} S_5\}$))

Potřebujeme rozumně definované rozklady, proto následující definice.

Plán p korektně implementuje operátor o (tj. je úplný a konzistentní plán pro dosáhnutí efektů o za předpokladu splněných PRECONDITIONS o) pokud:

1. p je konzistentní (tj. bez kontradikce v ORDERINGS i BINDINGS proměnných)
2. Každý efekt o je plněn aspoň jedním krokem p takovým, že není následně zrušen dalším krokem z p.
3. Každý předpoklad každého kroku v p musí být dosažen některým krokem v p nebo být obsažen v předpokladech o.

Pokud se nám povede takto definovat operátory dekompozice, tak se při provedení dekompozice nemusíme starat o vnitřní konzistenci plánu p. Musíme ale ověřit konzistenci plánu p s ostatními částmi plánu, protože tady můžou nastat konflikty.

Plánovač stačí mírně modifikovat – např. ke každému výběru kroku a cíle k hledání operátoru přidat další krok, který vybere abstraktní operátor a metodu dekompozice, kterou na něm provede.

Při dekompozici se mění celkový plán následovně:

- STEPS: odstraníme $S_{nonprim}$, přidáme všechny kroky plánu p.
- BINDINGS: Přidáme všechna přiřazení z plánu p. Pokud dojde ke sporu, selžeme a vynutíme backtracking.
- ORDERINGS: opět se chováme podle ”least commintment principle”. Nahradíme
  - všechny $S_a \prec S_{nonprim}$ podmínkami $S_a \prec S_{last}$, kde S_{last} reprezentuje poslední akci v p
  - všechny $S_{nonprim} \prec S_a$ podmínkami $S_{first} \prec S_a$, kde S_{first} reprezentuje první akce v p.

  Pak musíme zavolat RESOLVE–THREATS pro kontrolu dalších možných konfliktů.
- LINKS:

- $S_i \overrightarrow{c} S_{nonprim}$ nahradíme množinou hran $S_i \overrightarrow{c} S_m$, kde S_m je krok plánu p s c v předpokladech, který nemá přechůdce v p s c v předpokladech. Pokud takový S_m není, vazba byla zbytečná a můžeme jí zapomenout.
- Podobně $S_{nonprim} \overrightarrow{c} S_j$ nahradíme množinou hran $S_m \overrightarrow{c} S_j$, kde S_m je krok plánu p s efektem c, který nemá následníka v p s efektem c. Takový krok být musí, je–li dekompozice úplná.

Příklad.

Analýza hierarchické dekompozice

Dekompozice vypadá jako dobrý nápad. Jestli – a kolik – opravdu ušetří práce, to se snaží osvětlit následující rozbor.

Najít abstraktní řešení by nemělo být příliš složité. Jde spíš o to, aby to bylo užitečné, tedy abychom zamítáním nekonzistentních abstraktních řešení dělali něco užitečného – tj. zamítali nekonzistentní úplná řešení a nezamítali řešení konzistentní.

Proto bychom rádi, aby dekompozice splňovala:

- *upward solution property* – aby každá abstrakce konzistentního řešení byla konzistentní, tj. pokud najdeme nekonzistentní abstraktní řešení, můžeme ho zamítnout,
- *downward solution property* – aby každý konzistentní abstraktní plán byl abstrakcí nějakého úplného konzistentního řešení (obsahujícího pouze primitivní akce). Pak by nám stačilo

následovat konzistentní abstraktní řešení a backtrackovat jen na nejnižší – tj. nejméně abstraktní – hladině.

Redukce počtu "choose" bodů

Předpokládejme, že existuje aspoň jedno řešení o n základních krocích a že čas na RESOLVE THREATS a práci s podmínkami je zanedbatelný vzhledem k času na zpracování kroku plánovače. Nechť b je faktor větvení (branching faktor) dekompozice, každá dekompozice obsahuje s kroků plánu a počet hierarchických úrovní je d. Nehierarchické plánování potřebuje v nejhorším případě čas $O(b^n)$. Pro dekompozici s upward i downward solution property potřebujeme projít maximálně

$$\sum_{i=1}^{d} bs^i = O(bs^d)$$

kroků (na předposlední úrovni pro každý STEP d možností jak rozložit, krát počet úrovní).

Pro parametry $b = d = 3, s = 4, n = s^d = 64$ dostaneme 3×10^{30}

nehierarchicky versus 576 kroků hierarchicky.

Pokud neplatí upward a downward solution property a nemáme za ně rozumnou náhradu, pak nám hierarchické řešení zrychlení negarantuje (i když v průměrném chování může přinést).

Sdílení akcí v dekompozici

v dekompozici se pro každý operátor v p nedeterminitsticky rozhodneme, jestli pro něj využijeme některý z kroků *plan*u nebo pro něj vytvoříme novou instanci. Většinou je užitečná heuristika preferovat sdílení kroků, pokud to lze.

Mnohé hierarchické plánovače používají dekompozici bez sdílení a následně modifikují výsledný plán. Kritika (critics) je funkce, která bere na vstupu plán a vrátí plán s opravenými konflikty nebo–a dalšími anomáliemi.

Příklad: kompilátory na $\sin(x) + \cos(x)$ se chovají jako kritiky, tj. nesdílejí akce, protože by analýza možného sdílení trvala příliš dlouho.

Dekompozice versus aproximace

Místo (přesné a korektní) dekompozice můžeme hledat plán pomocí postupného zjemňování aproximace.

Každému předpokladu akce přiřadíme hladinu důležitosti (criticality level), např. pro akci `Buy`(x) srovnáme předpoklady od nejdůležitějšího:

1. Sells(store,x)
2. At(store)
3. Have(Money)

Pak nejdřív řešíme problém jen s předpoklady akcí důležitosti 1. Řešení nakoupí správné věci ve správných obchodech, ale nebude se starat o to, jak procházet mezi obchody a jak platit za zboží.

Až najdeme řešení na první hladině, rozšíříme ho o hladinu druhou,

a tak dále, dokud nenajdeme úplný a konzistentní plán.

Tento způsob aproximativního plánování můžeme reprezentovat i v HD–POP tak, že definujeme akci Buy_1 pouze z kritickými předpoklady, pro tuto akci definujeme dekompozici na Buy_2 s rozšířenými předpoklady. Pak už zbývá jen upravit pořadí prohledávání – nejdříve vyrobit celý plán bez dekompozic, pak teprve provést dekompozice. Při neúspěchu nejprve backtrackovat na hladině nejméně kritických premis.

V této dekompozici platí upward solution property, protože akce s vyššími čísly hladin vždy zdědí všechny premisy akcí předchozích, tj. máme–li řešení na hladině s vysokým číslem, máme řešení i na hladinách s nižšími čísly.

Toto přiřazení důležitosti předpokladů by se dalo dále zjemnit tím, že bychom dovolili hladinu důležitosti měnit podle kontextu – např. peníze se stanou daleko více kritické při koupi domu (ve srovnání s

koupí mléka).

Rozšíření jazyka akcí

- Podmíněné efekty
- Negované a disjunktivní cíle
- Univerzální kvantifikace
- Fluents (čas, peníze – spec. proměnné)

Podmíněné efekty

Už ve světě kostek jsme narazili na problém, že po položení kostky A na kostku B kostka B přestane být volná, ale po položení kostky A na stůl na stole stejně zbyde místo. Řešení pomocí dvou různých akcí a predikátů je krkolomné a neefektivní, lepší je rozlišit efekt akce podle toho, jestli kostku pokládáme na jinou kostku či na stůl.

akce	PRECONDITIONS	DELETE
STACK(x,y)	CLEAR(y) & HOLDING(x)	(CLEAR(y) **when** $y \neq \mathit{Table}$) & HOLDING

Máme–li podmíněný efekt, tak podmínku tohoto efektu vybíráme za cíl pouze v případě, že efekt potřebuje nějaká kauzální hrana.

Také musíme odstranit potenciální hrozby. Jedna z možností je tzv. konfrontace – pokud hranu $S_i \overrightarrow{c} S_j$ ohrožuje podmíněný efekt ($\neg c$ when p), tak zajistíme, aby p nebylo splněno – např. dosazením

$y = \mathit{Table}$ ve výše uvedeném příkladu.

Negované a disjunktivní cíle

Konfrontace výše nám zavedla negovaný cíl $\neg p$. To nám ale nezvyšuje složitost – opět jen projdeme efekty akcí a vybereme ty, které cíl obsahují. Jen musíme dát pozor, aby unifikace spojila p a $\neg\neg p$. Také musíme zavést speciální pravidlo pro počáteční stav, kde negativní vlastnosti nespecifikujeme přímo, ale předpokládáme, že platí negace všeho, co není řečeno pozitivně.

Ještě snadněji můžeme přidat disjunktivní předpoklady – prostě nedeterministicky vybereme jeden jako podcíl, až přijde čas.

Horší je to s disjunktivními efekty, protože nám mění deterministické prostředí na nedeterministické.

Univerzální kvantifikace

Ve světě kostek jsme museli zavést predikát $Clear(B)$. Lepší by bylo napsat přímo do předpokladů operátoru definici predikátu $Clear(B)$, tj. $\forall x(Block(x) \rightarrow \neg On(x, b))$. To nám dovolí nejen zrušit udržování predikátu $Clear(B)$, ale také plánování ve složitějších světech, např. kostkami různých velikostí.

Podobně můžeme zavést univerzální kvantifikátor do efektů. Například přenesením batohu přeneseme i všechny předměty uvnitř batohu, tj. efekt Carry(Bag, x,y) je
$\forall i(Item(i) \rightarrow (\neg At(i, x) \& At(i, y) \text{when} In(i, bag)))$.

I když to vypadá jako univerzální kvantifikace, udržíme svět omezený tím, že na začátku budou dané typy věcí a budeme znát všechny předměty daného typu. Tím pak budeme moci nahradit univerzální kvantifikaci výčtem všech prvků daného typu. Není to

příliš efektivní, ale není obecné lepší řešení.

Univerzálně kvantifikované efekty nemusíme expandovat, protože nám na většině efektů nezáleží – musíme jen zajistit, aby si jich všiml RESOLVE–THREATS jako možných hrozeb a CHOOSE–OPERATOR jako možných ″ocasů″ kauzálních hran.

Pozn: možnost nahradit detailního typu predikáty (Actual(House1)), v případě velkého počtu nerozlišitelných objektů je brát jako zdroje – resources – např. peníze, čas, lidské síly.

Pozn: Russel–Norvig – POP–DUNC – algoritmus, kde je vše dohromady.

Omezení na zdroje jako čas a lidé

- Fluents – předem definované proměnné, specielní zacházení
- Hrubý odhad – dolní a horní hranice, odhad ceny za jednotku atd.
- v předpokladech testy na nerovnosti s fluents
- v efektech aritmetický výpočet nové hodnoty fluent ze staré hodnoty fluent

Je otázkou, nakolik ověřovat konzistenci – většinou se hlídá jen na hrubo, že akce samostatně potřebuje jen zdroje, které jsou dosažitelné. Šlo by i řešit constraint satisfaction problem, ale to by mohlo trvat moc dlouho na to, jak často to potřebujeme dělat.

Čas je skoro jako jiné fluent, ale

- nelze poslat pozpátku (i když třeba nádrž benzínu můžeme

doplnit).

- Navíc čas musí korespondovat s uspořádáním $\prec$.