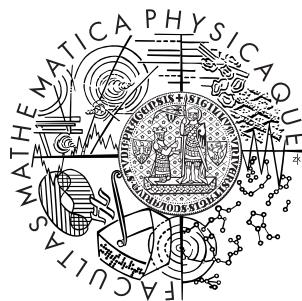


Univerzita Karlova v Prahe  
Matematicko-fyzikálna fakulta

## BAKALÁRSKA PRÁCA



Róbert Sasák

### Klasické plánovacie techniky

Katedra teoretickej informatiky a matematickej logiky

Vedúci bakalárskej práce: doc. RNDr. Roman Barták, Ph.D.,  
Štúdijný program: všeobecná informatika

2009



Ďakujem doc. RNDr. Romanovi Bartákovi, Ph.D. za odborné vedenie mojej práce, jeho čas, trpezlivosť, konštruktívne pripomienky a doporučenia, ktoré mi venoval.

Prehlasujem, že som svoju bakalársku prácu napísal samostatne a výhradne s použitím citovaných prameňov. Súhlasím so zapožičaním práce a jej zverejnením.

V Prahe dňa 9.12.2009

Róbert Sasák



# Obsah

<b>1</b>	<b>Úvod</b>	<b>9</b>
<b>2</b>	<b>Plánovanie</b>	<b>11</b>
2.1	Klasické plánovanie . . . . .	12
2.2	Reprezentácia stavu . . . . .	13
2.3	Plánovací problém . . . . .	14
<b>3</b>	<b>Prehľad klasických techník</b>	<b>15</b>
3.1	Dopredné plánovanie . . . . .	15
3.2	Spätné plánovanie . . . . .	16
3.2.1	Liftovaná verzia spätného plánovania . . . . .	17
3.2.2	Zakázane stavy . . . . .	18
<b>4</b>	<b>Implementácia</b>	<b>21</b>
4.1	Parser . . . . .	21
4.2	Plánovače . . . . .	22
4.2.1	Spôsob prehľadávania . . . . .	22
4.2.2	Heuristika . . . . .	25
4.2.3	Moduly . . . . .	27
<b>5</b>	<b>Porovnanie</b>	<b>29</b>
5.1	Blocks World . . . . .	29
5.2	Elevators . . . . .	31
5.3	Gripper . . . . .	32
5.4	Hanoi . . . . .	33
<b>6</b>	<b>Záver</b>	<b>35</b>
	<b>Literatúra</b>	<b>38</b>
<b>A</b>	<b>Domény</b>	<b>39</b>
<b>B</b>	<b>Planning Domain Definition Language (PDDL)</b>	<b>43</b>
B.1	Definícia domény . . . . .	43
B.2	Definícia problému . . . . .	45
B.3	História . . . . .	46
<b>C</b>	<b>Výsledky</b>	<b>49</b>



Názov práce: Klasické plánovacie techniky

Autor: Róbert Sasák

Katedra: Katedra teoretickej informatiky a matematickej logiky

Vedúci bakalárskej práce: doc. RNDr. Roman Barták, Ph.D.

[bartak@kti.mff.cuni.cz](mailto:bartak@kti.mff.cuni.cz)

Abstrakt: Klasické plánovanie sa zaobera hľadaním postupnosť akcií, ktoré prevádzajú počiatočný stav sveta na požadovaný konečný stav. Predložená práca pojednáva o dvoch metódach klasického plánovania: doprednom a spätnom plánovaní. Obe metódy sme implementovali formou softvérového prototypu využitím piatich prehľadávaní: DFS, BFS, IDDFS, A\*, WA\*. Ďalším rozšírením o viaceré heuristiky sme získali celkovo 26 plánovačov. Efektivitu plánovačov sme porovnali na niekoľkých doménach z medzinárodnej plánovaciej súťaže. Žiadnen z plánovačov nie je výrazne lepší na všetkých doménach, ale vo všeobecnosti boli lepšie dopredné plánovače.

Kľúčové slová: klasické plánovanie, dopredné plánovanie, spätné plánovanie

Title: Classical planning techniques

Author: Róbert Sasák

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: doc. RNDr. Roman Barták, Ph.D.

Supervisor's e-mail address: [bartak@kti.mff.cuni.cz](mailto:bartak@kti.mff.cuni.cz)

Abstract: Classical planning deals with finding a sequence of actions transferring the initial state of world into a desired goal state. This work surveys two classical planning techniques, forward and backward search. We implement both techniques in a form of software prototype using five different search algorithms, in particular DFS, BFS, IDDFS, A\*, WA\*. By introducing additional heuristic we get family of 26 planners. We compare effectivity of the planners on several domains from International Planning Competition. None of the planners is significantly better on all domains, however, in general, the planners based on forward search perform better.

Keywords: classical planning, forward search, backward search



# Kapitola 1

## Úvod

Kedykoľvek sa zaujímame o poradie a postupnosť krokov v nejakej činnosti, tak sa venujeme plánovaniu. Plánovanie je typická činnosť pre ľudí, avšak nie pre stroje, a preto tvorí rozsiahlu oblasť v štúdiu umelej inteligencie. Plánovacie techniky sú kľúčové na riešenie rôznych úloh od robotiky, plánovania procesov, zhromažďovania informácií, autonómnych agentov až po riadenie vesmírnych družíc.

Každé dva roky sa koná medzinárodná plánovacia súťaž (IPC) v rámci medzinárodnej konferencie v plánovaní a rozvrhovaní (ICAPS). V rámci tejto súťaže medzi sebou súťažia programy nazývané plánovače. Zjednodušenie by sa sa dalo povedať, že cieľom plánovača je nájsť najlepší plán pre daný konkrétny problém a to v najkratšom čase. Rôzne plánovače používajú rôzne komplexné algoritmy na nájdenie plánu. Keďže rýchlosť plánovača je okrem algoritmu aj úzko spätá s konkrétnou implementáciou, je ľahšie len z porovnania výsledkov súťaže určiť najvhodnejší algoritmus.

Zjednotením implementácie rôznych algoritmov získame relevantný systém pre porovnanie a určenie ich reálnej efektivity. Výsledky z takéhoto systému budú smerodajné pre ďalší vývoj algoritmov alebo ich rozšírenie.

## Ciel' práce

Cieľom práce je spracovať prehľad klasických plánovacích techník ako je dopredné a spätné plánovanie a tieto techniky implementovať formou softvérového prototypu s jednotným rozhraním a porovnať ich efektivitu.

Ďalej si v tejto práci kladieme za cieľ porovnať implementované plánovače na štandardných plánovacích problémoch.

## Štruktúra práce

Nasledujúca kapitola nazvaná Plánovanie predstavuje vstupnú bránu do problematiky. Zavádza terminológiu použitú v tejto práci, pričom sa od plánovania ako takého upriamuje na klasické plánovanie.

Tretia kapitola bližšie vysvetľuje klasické plánovacie techniky ako sú dopredné a spätné plánovanie, ktoré sú jadrom celej práce. Spätné plánovanie je následne rozšírené o liftovanú verziu. Na konci kapitoly je časť zaoberajúca sa zefektívnením spätného plánovania, pomocou detekcie zakázaných stavov.

Po teoretickom úvode nasleduje kapitola, ktorá sa zaoberá praktickou implementáciou plánovačov v jazyku Prolog. Sústredí sa na problémy, ktoré vznikli pri vlastnej implementácii plánovača. Kapitola je tématicky rozdelená na časť opisujúcu implementáciu parsera jazyka PDDL a plánovačov.

Štvrtá kapitola zavádzá metodiku porovnania plánovačov a prezentuje získané výsledky. S touto kapitolou taktiež súvisí dodatok C, ktorý obsahuje podrobnejšie výsledky experimentov pre jednotlivé testované domény.

Na lepšie vysvetlenie niektorých problémov v plánovaní sú použité jednoduché príklady. Z nich sa väčšina v tejto práci vzťahuje k najznámejšej doméne **Blocks World**, ktorá je bližšie popísaná v dodatku A. Ďalej sú v dodatku A stručne spomenuté aj ďalšie domény, ktoré boli použité na testovanie.

V dodatku B je predstavený jazyk PDDL, ktorý sa používa na zápis plánovacích problémov.

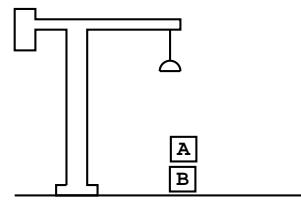
Pred začatím čítania práce by mal byť čitateľ oboznámený s doménou **Blocks World** a jazykom PDDL. Kapitoly 2 až 6 na seba postupne nadväzujú.

# Kapitola 2

## Plánovanie

V tejto kapitole zavedieme pojmy používané v tejto práci, z ktorých väčšina bola prevzatá z [13].

Plánovanie je proces výberu a usporiadanie akcií k dosiahnutiu určitého cieľa. Za akciu môžeme poklať ľubovoľnú činnosť ako napr. obutie topánok, otvorenie dverí alebo naloženie tovaru. Zoznam akcií nazývame plán. Vykonaním akcie nastane zmena - akcia zmení stav systému. Prirodzený spôsob ako popísat stav je pomocou vlastnosti.



Obr. 2.1: Príklad stavu z domény Blocks World

**Príklad 2.0.1.** Na obrázku 2.1 je príklad stavu pre doménu *Blocks World*. Tento stav môžeme popísať nasledujúcimi vlastnosťami: ruka je prázdna, kocka A leží na kocke B, kocka B leží na stole a na kocke A nie žiadna iná kocka. Tieto vlastnosti plne špecifikujú stav na obrázku. V definícii 2.2.1 bude stav zavedený formálne.

Cieľ môže byť špecifikovaný viacerými spôsobmi:

- Ako cieľový stav tzn. presný popis ako má vyzeráť systém. V prípade, že existuje viacero cieľových stavov, tak je cieľ špecifikovaný ako množina cieľových stavov.
- V niektorých prípadoch je cieľový stav príliš špecifický, hlavne keď nás zaujímajú iba jeho niektoré vlastnosti. Napríklad ak plánujeme prevoz tovaru z miesta A na miesto B, tak nie je podstatné akým autom sa tovar presunul a kde je auto zaparkované. Dôležité je, že tovar sa nachádza na mieste B.

Preto môže byť výhodou špecifikovať cieľ ako čiastočný popis stavu.

- Predošlé špecifikácie môžu byť ešte doplnené o podmienky, ktoré musia byť splnené pre všetky stavy, ktorými prechádza výsledný plán. Príklad podmienky môže byť, že traktor nemôže ísť po diaľnici.

Počiatočný stav spolu so zoznamom akcií generujú stavový priestor. Stavový priestor je orientovaný graf, ktorého vrcholy sú jednotlivé stavy a hrany

tvoria akcie. Dva stavy sú spojené hranou ak existuje akcia, ktorá mení jeden stav na druhý. Tým, že stavový priestor nie je zadaný explicitne, môžeme popísat veľmi veľký stavový priestor. Na druhú stranu však nie je možné použiť na takto reprezentovanom stavovom priestore jednoduché grafové algoritmy na hľadanie najkratšej cesty.

Plánovač nazývame program, ktorý pre zadaný počiatočný stav, dostupné akcie a cieľ, určí poradie akcií (plán), ktoré povedú od počiatočného stavu k cieľu.

Samozrejme takýchto plánov môže byť veľa a je preto vhodné zaviesť metriku, ktorá určí optimálny plán. Príkladom metrík môže byť:

- Dĺžka plánu. Optimálny plán obsahuje minimálny počet akcií k dosiahnutiu cieľa.
- Čas trvania plánu (Ak akcie majú zadanú dĺžku trvania). Optimálny plán bude trvať najkratšie.

Alebo môže metrika kombinovať oba spomínané príklady. Vo všeobecnosti môže metrika minimalizovať alebo maximalizovať ľubovoľnú funkciu z parametrov plánu.

## 2.1 Klasické plánovanie

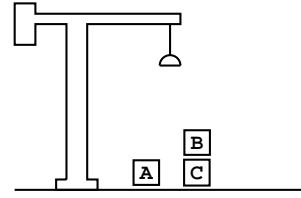
V predošej časti sme zaviedli pojem plánovanie. Pre navrhnutie plánovača je nutné plánovanie formalizovať tzn. zjednodušiť plánovanie ako také, ale aby stále pokrývalo zaujímavé plánovacie problémy z reálneho sveta. A práve klasické plánovanie predstavuje teoretický model, ktorý zjednodušuje plánovanie nasledovne:

- Predpokladáme deterministický svet tzn. aplikovaním akcie na stav získame práve jeden nový stav.
- Predpokladáme konečný stavový priestor. Počet stavov, v ktorých sa môže nachádzať systém, je obmedzený.
- Svet a jeho akcie sú pre plánovač vopred známe. Opakom môže byť plánovač, ktorý plánuje pohyb robota v neznámej krajine. Robot pozná iba časť stavového priestoru, ktorý zodpovedá vzdialosti pokiaľ robot dovidí.
- Svet je nemenný počas celého plánovania. Hovoríme aj o Offline plánovaní.
- Všetky akcie trvajú rovnako dlho. V tomto modely sa neuvažuje čas.

Tento model predstavuje základ pre ďalšie študovanie plánovania, pričom mnohé jeho vlastnosti zostávajú rovnaké aj v modeloch bez hore uvedených obmedzení.

## 2.2 Reprezentácia stavu

V klasickom plánovaní sa používajú tri druhy reprezentácií stavu: množinová reprezentácia, klasická reprezentácia a reprezentácia pomocou stavových premenných. Všetky tri reprezentácie majú rovnakú vyjadrovaciu hodnotu, ale klasická reprezentácia je pre svoju prehľadnosť najpoužívanejšia. Túto reprezentáciu budeme používať v celej práci.



Obr. 2.2:

**Definícia 2.2.1.** (Stav) *Stav je reprezentovaný ako množina inštanciovaných predikátov, kde pod predikátom si zjednodušene môžeme predstaviť vlastnosť sveta.*

**Príklad 2.2.1.** Na obrázku 2.2 je príklad stavu zo sveta kociek. Príklad inštanciovania predikátu uvedieme na predikáte `ontable(x)`. Dosadením konštanty `A` za premennú `x` získame inšanciu predikátu `ontable(A)`. Situácia na obrázku je popísaná stavom  $s = \{\text{ontable}(A), \text{clear}(A), \text{ontable}(B), \text{on}(B, C), \text{clear}(B), \text{handempty}\}$ .

**Definícia 2.2.2.** (Ciel) *V klasickej reprezentácii je ciel' g reprezentovaný ako množina plne inštanciovaných predikátov. Stav s nazývame cielový, ak sú všetky predikáty ciela g splnené v stave s tzn.  $g \subseteq s$ .*

**Definícia 2.2.3.** (Operátor) *Operátor je označený menom a parametrami, a pozostáva z troch množín: predpoklady, pozitívne efekty a negatívne efekty, označené ako  $\text{precond}(o)$ ,  $\text{effects}^+(o)$  a  $\text{effects}^-(o)$ . Všetky tri množiny obsahujú neinštanciované predikáty. Parametre tvorí zoznam voľných premenných z troch spomenutých množín. Množinu všetkých operátorov značíme  $O$ .*

**Definícia 2.2.4.** (Akcia) *Inštanciovaný operátor sa nazýva akcia tzn. že za všetky voľné premenné sú dosadené konštanty. Analogicky k množine všetkých operátorov  $O$  definujme množinu všetkých akcií  $A$ .*

$$A = \{a | o \in O \cup a \text{ is ground instance of } o\}$$

**Príklad 2.2.2.** Ako príklad si zoberieme operátor `o=stack(x,y)`, ktorý vo svete kociek odpovedá položeniu kocky `x` na kocku `y`. Formálne je definovaný nasledovne:

- $\text{precond}(o) = \{\text{holding}(x), \text{clear}(y)\}$
- $\text{effects}^+(o) = \{\text{clear}(x), \text{handempty}\}$
- $\text{effects}^-(o) = \{\text{holding}(x), \text{clear}(y)\}$

*Inštanciu operátora (akciu) a získame dosadením konkrétnych objektov za premenné `x` a `y`.*

Akcia  $a$  je aplikovateľná na stav  $s$  ak sú splnené jej predpoklady, teda  $\text{precond}(a) \subseteq s$ . Aplikovaním akcie získame nový stav:

$$s' = \gamma(s, a) = (s - \text{effects}^-(a)) \cup \text{effects}^+(a)$$

Kde  $\gamma$  sa nazýva prechodová funkcia. Hovoríme, že akcia  $a$  je relevantná pre cieľ  $g$ , ak približuje stav k cieľovému stavu ( $g \cap \text{effect}^+(a) \neq \emptyset$ ) a zároveň nie je v konflikte ( $g \cap \text{effect}^-(a) = \emptyset$ ). Pre všetky relevantné akcie definujeme inverznú prechodovú funkciu:

$$\gamma^{-1}(g, a) = (g - \text{effects}(a)) \cup \text{precond}(a)$$

## 2.3 Plánovací problém

Na záver uvedieme definíciu plánovacieho problému. Plánovací problem  $\mathcal{P}$  je trojica  $(O, s_0, g)$ , kde  $O$  je množina operátorov,  $s_0$  je počiatočný stav a  $g$  je cieľ.

Formálne sa plánovací problém ešte delí na doménu a problém. Doméne zodpovedá popis sveta tzn. zoznam operátorov, ktorý presne popisuje dynamiku vo svete. Problém predstavuje konkrétnu situáciu vo svete, ktorú chceme vyriešiť. Problému zodpovedá počiatočný stav a cieľ.

Riešenie plánovacieho problému je zoznam akcií (plán)  $a_1, \dots, a_k$ , ktorých postupným aplikovaním získame stav  $s_g$  splňujúci cieľ  $g$ :

$$g \subseteq s_g = \gamma(\dots \gamma(s_0, a_1), \dots, a_k)$$

Nás bude zaujímať najkratší plán čo do počtu akcií - optimálny plán.

# Kapitola 3

## Prehľad klasických techník

Na riešenie klasických plánovacích problémov existujú dva prístupy. Obidva sú založené na prehľadávaní. Prvý, intuitívny prístup, prehľadáva priestor všetkých stavov a druhý, zložitejší prístup, prehľadáva priestor všetkých plánov. V tejto kapitole uvedieme najznámejšie techniky prehľadávania priestoru stavov, na ktoré je táto práca zameraná.

### 3.1 Dopredné plánovanie

Dopredné plánovanie (Forward-Search) predstavuje prirodzenú metódu na riešenie plánovacieho problému. Postupne rozvíja čiastočný plán, až kým nedosiahne cieľový stav. Algoritmus začína s počiatočným stavom  $s_0$ . Následne sa nedeterministicky vyberie aplikovateľná akcia  $a$ . Aplikovaním akcie sa získa nový stav. Postup sa opakuje, pokiaľ nie je splnený cieľ  $g$ . Pseudokód algoritmu je uvedený v algoritme 1.

---

**Algoritmus 1** Algoritmus dopredného plánovania

---

```
procedure forward-search( $O, s_0, g$ )
     $s \leftarrow s_0$ 
     $\pi = \emptyset$ 
    while  $g \not\subseteq s$  do
         $applicable \leftarrow \{a | o \in O \wedge a \text{ is ground instance of } o \wedge \text{precond}(a) \subseteq s\}$ 
        if  $applicable = \emptyset$  then
            print "Goal is not reachable from init state."
            return failure
        end if
         $a \in applicable$  {Nondeterministically choose an action  $a$ }
         $s \leftarrow \gamma(s, a)$ 
         $\pi \leftarrow \pi.a$ 
    end while
    return  $\pi$ 
end procedure
```

---

**Príklad 3.1.1.** Predpokladajme situáciu na obrázku 2.2 a akciu  $\text{pick-up}(B)$ . Pred aplikovaním akcie musia byť splnené predpoklady akcie:

$$\begin{aligned} \text{precond}(a) &\subseteq s \\ \{\text{handempty}, \text{clear}(B)\} &\subseteq \{\text{ontable}(A), \text{clear}(A), \text{ontable}(C), \text{on}(B, C), \\ &\quad \text{clear}(B), \text{handempty}\} \end{aligned}$$

A následne sa môže akcia aplikovať a tým získať nový stav  $s'$ :

$$\begin{aligned} \gamma(s, a) &= (s - \text{effects}^-(a)) \cup \text{effects}^+(a) \\ &= \{\text{ontable}(A), \text{clear}(A), \text{ontable}(C), \text{clear}(C), \text{holding}(B)\} \end{aligned}$$

Dopredné plánovanie je možné implementovať deterministicky napríklad prehľadávaním do hĺbky. Tejto problematike sa budeme venovať podrobnejšie v časti Implementácia.

## 3.2 Spätné plánovanie

Počiatočný stav úplne popisuje stav domény. Naopak cieľ zvyčajne špecifikuje koncový stav iba čiastočne, a preto býva omnoho menší (množinovo) ako počiatočný stav. Túto myšlienku využíva algoritmus spätného plánovania. Algoritmus sa začína s cieľom. Spätným aplikovaním akcie získa podcieľ. Ak podcieľ zahrnuje počiatočný stav, tak algoritmus našiel postupnosť spätných akcií. Ak nie, algoritmus zoberie za svoj cieľ aktuálny podcieľ a hľadá pre nový cieľ opäťovne nový podcieľ. Pripomenieme, že pre daný cieľ  $g$  a akciu  $a$  môžme získať podcieľ  $g'$  nasledovne:

$$g' = \gamma^{-1}(g, a) = (g - \text{effects}(a)) \cup \text{precond}(a)$$

Pseudokód algoritmu je uvedený v algoritme 2.

---

### Algoritmus 2 Algoritmus spätného plánovania

---

```

procedure backward-search( $O, s_0, g$ )
 $\pi = \emptyset$ 
while  $g \not\subseteq s_0$  do
     $relevant \leftarrow \{a | o \in O \wedge a \text{ is ground instance of } o \wedge a \text{ is relevant}\}$ 
    if  $relevant = \emptyset$  then
        print "Goal is not reachable from init state."
        return failure
    end if
     $a \in relevant$  {Nondeterministically choose an action  $a$ }
     $g \leftarrow \gamma^{-1}(g, a)$ 
     $\pi \leftarrow a.\pi$ 
end while
return  $\pi$ 
end procedure

```

---

**Príklad 3.2.1.** Zoberme si ako príklad doménu *Blocks World*. Predpokladajme cieľ  $g = \{on(A, B), on(B, C)\}$  a chceme získať podcieľ  $g'$  spätným aplikovaním akcie  $a$ . Z definície relevantnosti akcie je vidieť, že sa môže jednať napríklad o akciu  $\text{stack}(A, B)$ . Aplikovaním inverznej prechodovej funkcie získame podcieľ  $g'$ :

$$\begin{aligned}\gamma'(g, a) &= (g - \text{effects}(a)) \cup \text{precond}(a) \\ &= \{\text{holding}(A), \text{clear}(B), \text{on}(B, C)\}\end{aligned}$$

Akcia  $\text{stack}(B, C)$  je tiež podľa definície relevantná. Podobným spôsobom ako pre akciu  $\text{stack}(A, B)$  by sme získali:

$$\{\text{holding}(B), \text{clear}(C), \text{on}(A, B)\}$$

Ale na kocke  $B$  leží kocka  $A$ ! Samozrejme tento cieľ nezodpovedá realite.

Tak ako bolo uvedené v príklade, z dostupných informácií o doméne nie je možné priamo rozoznať stavy, ktoré nezodpovedajú realite. Pre konkrétnu doménu *Blocks World* je možné spísať niekoľko pravidiel, ktoré musí ľubovoľný stav (alebo podcieľ) splňať. Napríklad kocka  $A$  nemôže ležať na samej sebe, čomu zodpovedá zakázaný predikát  $\text{on}(A, A)$ .

Vo všeobecnosti pre ľubovoľnú doménu, by získanie všetkých pravidiel znamenalo prejdenie celého stavového priestoru, čo by prevýšilo vyriešenie plánovacieho problému[10].

Algoritmus je možné, tak ako pri doprednom plánovaní, implementovať deterministicky. Tejto problematike sa budeme venovať podrobnejšie v časti Implementácia.

### 3.2.1 Liftovaná verzia spätného plánovania

**Príklad 3.2.2.** Predpokladajme doménu podobnú doméne *Blocks World*, ktorá obsahuje viac rúk označených  $H1, \dots, H9$ . A nech koncový stav je  $\text{on}(A, B)$ . Na takto definovanom plánovacom probléme spustíme spätné prehľadávanie. Algoritmus správne určí operátor  $\text{stack}(x, A, B)$ , avšak nevie čo má dosadiť za premennú (ruku)  $x$ . Preto musí vyskúšať všetky možnosti:  $\text{stack}(H1, A, B)$ , ...,  $\text{stack}(H9, A, B)$ . Pritom nie je dôležité, ktorá ruka držala kocku  $A$ .

Pri spätnom prehľadávaní je často známy iba jeden parameter operátora. Zvyšné neznáme parametre sa musia postupne skúsať. To zvyšuje vetviaci faktor.

Liftovaná verzia spätného prehľadávania rieši tento problém. Hlavnou myšlienkovou je odložiť inštanciovanie premennej na neskôr. Volné premenné sa zuniifikujú buď v nasledujúcich akciách alebo pri dosiahnutí počiatočného stavu. Pseudokód algoritmu je uvedený v algoritme 3.

Pretože podciele nie sú plne inštanciované, je detekcia cyklov v deterministickej implementácii zložitejšia. Zoberme si ako príklad prehľadávanie do hĺbky. Nech postupnosť už navštívených podcieľov je  $(g_1, \dots, g_k)$ , kde  $g_k$  je posledne navštívený podcieľ. Cyklus nastane ak existuje  $i < k$  také, že  $g_i$  sa zuniifikuje s podmnožinou  $g_k$ .

---

**Algoritmus 3** Liftovaná verzia spätného plánovania

---

```
procedure lifted-backward-search( $O, s_0, g$ )
 $\pi = \emptyset$ 
while  $g \not\subseteq s_0$  do
     $relevant \leftarrow \{ (o, \sigma) | o \in O \wedge o \text{ is relevant} \wedge \sigma_1 \text{ is an substitution that}$ 
     $\text{standartizes } o \text{'s variables, } \sigma_2 \text{ is a most general unifier for } \sigma_1(o) \text{ and the}$ 
     $\text{atom of } g \text{ that } o \text{ is relevant for, and } \sigma = \sigma_1\sigma_2 \}$ 
    if  $relevant = \emptyset$  then
        print "Goal is not reachable from init state."
        return failure
    end if
     $(o, \sigma) \in relevant \{ \text{Nondeterministically choose a pair } (o, \sigma) \}$ 
     $g \leftarrow \gamma^{-1}(\sigma(g), \sigma(o))$ 
     $\pi \leftarrow \sigma(o).\sigma(\pi)$ 
end while
return  $\pi$ 
end procedure
```

---

### 3.2.2 Zakázane stavby

Ako bolo ukázané v príklade 3.2.1 pri spätnom prehľadávaní sa prehľadávajú aj stavby, ktoré nepatria do stavového priestoru plánovacieho problému (zakázané stavby). Ak by sme vedeli rozlísiť tieto stavby, tak by sa zmenšil vetviaci faktor prehľadávania. Ako už bolo spomenuté, samotná detekcia týchto stavov je komplikovanejšia ako nájdenie plánu.

Preto uvedieme metódu, ktorá dokáže rozoznať aspoň časť zo zakázaných stavov. Metóda spočíva v nájdení dvojíc predikátov, ktoré ak sú splnené v danom stave, tak sa jedná o zakázaný stav. Pojmy v tejto kapitole boli prevzaté z [11].

**Definícia 3.2.1.** (Zakázané stavby) Majme plánovací problém  $\mathcal{P} = (O, s_0, g)$ .  $M$  je množina zakázaných dvojíc predikátov, práve keď pre všetky dvojice  $R = \{p, q\} \in M$  platí:

1.  $p$  a  $q$  nie sú splnené v  $s_0$ .
2. Pre všetky operátory  $o \in O$ , pre ktoré  $p \in effects^+(o)$ , predikát  $q$  je bud'  $q \in effects^-(o)$  alebo  $q \notin effects^+(o)$  a existuje predpoklad  $r \in precond(o)$ , že pári  $\{r, q\} \in M$ .

Prvá podmienka je jednoduchá. Ďalej ak všetky operátory, ktoré pridávajú predikát  $p$  zároveň odoberajú predikát  $q$ , tak sa nemôžu predikáty  $p$  a  $q$  vyskytovať spoločne v jednom stave. Druhá časť druhej podmienky tranzitívne pridáva zakázaný vzťah na základe nezlučiteľnosti predpokladu.

Z definície 3.2.1 nie je jasné ako by sa dala množina  $M$  postupne vygenerovať. Začneme preto s množinou všetkých párov predikátov  $M_0$  a z tejto množiny postupne zmažeme všetky dvojice, ktoré nespĺňajú prvú a druhú podmienku definície. Takto získame množinu  $M$ , ktorá splňuje definíciu 3.2.1.

Do množiny  $M_0$  pritom nie je nutné zobrať všetky dvojice, ale postačí zobrať:

- Všetky dvojice  $\{p, q\}$ , pre ktoré existuje akcia, ktorá pridáva  $p$  a maže  $q$ .
- A ďalej všetky dvojice  $\{r, q\}$ , pre ktoré existuje zakázaný páár  $\{p, q\}$  a existuje operátor  $o$ , pritom predikát  $r \in \text{precond}(o)$  a  $p \in \text{effects}^+(o)$ .

Pseudokód algoritmu na hľadanie zakázaných dvojíc je uvedený v algoritme 4.

---

**Algoritmus 4** Hľadanie navzájom zakázaných párov predikátov

---

```

procedure mutexes( $O, s_0$ )
 $M_A \leftarrow \{\{p, q\} | \exists o \in O : p \in \text{effects}^+(o) \wedge q \in \text{effects}^-(o)\}$ 
 $M_B \leftarrow \{\{r, q\} | \exists p : \{p, q\} \in M_A \exists r \in \text{precond}(o) \wedge p \in \text{effects}^+(o)\}$ 
 $M_0 = M_A \cup M_B$ 
 $M = \emptyset$ 
for  $\{p, q\} \in M_0$  do
    if  $p \notin s_0 \vee q \notin s_0$  then {1.podmienka}
         $M \leftarrow M \cup \{\{p, q\}\}$ 
    end if
end for
while  $\exists(p, q) \in M$  nespĺňa 2.podmienku do
    delete  $(p, q)$  from  $M$ 
end while
return  $M$ 
end procedure

```

---

Táto metóda sa dá ďalej zovšeobecniť na hľadanie trojíc, štvoric, atď. Množstvo nájdených zakázaných n-tíc však nie je nijak predvídateľné a závisí na konkrétnej doméne.



# Kapitola 4

## Implementácia

V predchádzajúcich kapitolách sme predstavili plánovanie a dve klasické techniky na hľadanie plánu. Pri klasickom plánovaní zostaneme, avšak v tejto kapitole sa zameráme na implementáciu už spomenutých algoritmov v praxi. Zameriame sa na problémy, ktoré je potrebné vyriešiť všeobecne, bez ohľadu na konkrétnu implementáciu v programovacom jazyku. Výnimku v tomto bude tvoriť časť, ktorá opisuje implementáciu parséra.

### Výber programovacieho jazyka

My sme sa rozhodli pre implementáciu v neprocedurálnom programovacom jazyku Prolog a to z nasledujúcich dôvodov. Prolog patrí medzi deklaratívne programovacie jazyky[9] tzn. že programátor popisuje iba cieľ, pričom presný postup výpočtu je ponechaný na systéme. Prolog disponuje tromi základnými vstavanými schopnosťami: unifikácia, rekurzia a backtracking. Posledné dve spomenuté, sa dajú s výhodou využiť na prehľadávanie stavového priestoru. V neposlednom rade to bola výzva napísat parser pre jazyk PDDL [príloha B] v Prologu.

Existuje viacero implementácií Prologu, ktoré sa hlavne líšia prídavnými knižnicami. My sme sa rozhodli pre komerčnú verziu SICStus Prolog.

### Logické rozdelenie

Z implementačného hľadiska je možné prácu rozdeliť na časť zabezpečujúcu spracovanie vstupu (Parser) a časť, ktorá rieši plánovací problém (Solver). V časti parser je popísaná implementácia načítania plánovacieho problému zo súboru a za ňou nasledujú implementácie rôznych techník na riešenie plánovacieho problému.

### 4.1 Parser

Na zápis plánovacích problémov sa používa jazyk PDDL, ktorý je bližšie opísaný v dodatku B. Tak ako už bolo spomenuté v časti 2.3, plánovací problem je rozdelený na dve časti: doména a problém. Doména špecifikuje prob-

lém abstraktne na úrovni dostupných objektov, akcií a funkcií. Zjednodušene by sa dalo povedať, že definuje všetko to, čo je nemenné počas plánovania. Naopak plánovací problém je inštancia domény, ktorý vymenúva začiatokný stav, cieľ, konkrétné objekty a hodnoty pre funkcie. A teda dynamické prostriedky plánovacieho problému. Rozdelenie plánovacieho problému na doménu existuje z historického hľadiska. Prvé plánovače boli navrhnuté pre jednu danú konkrétnu doménu.

Syntax jazyka PDDL je odlišná od prologovských termov. Každá klauzula v PDDL je zapísaná v zátvorkách a nie je ju možné načítať ako prologovský term. Naskytla sa otázka ako previesť formát jazyka PDDL do formátu vhodného pre prácu v Prologu. V čase písania práce nie je dostupné žiadne verejné riešenie. Jazyk PDDL je v špecifikácii [12] presne popísaný pomocou BNF (Backus–Naurovej Formy), čo je iný zápis bezkontextových gramatik. Prolog obsahuje jednoduchý systém operátorov na prácu s gramatikami DCG (Definite Clause Grammar) [8].

Jednoduchými úpravami zväčša technického typu je možné prepísať riadok po riadku BNF špecifikáciu do DCG gramatiky. Pričom sa zachováva prehľadnosť BNF špecifikácie.

**Príklad 4.1.1.** *Priklad prepisu BFF špecifikácie do DCG gramatiky. Prvý riadok zodpovedá BNF a v druhom riadku je prepis do DCG.*

```
<types-def> ::= (:types           <typed_list(name)>)
types_def(L) --> [',':',types], typed_list(name, L), [')'].
```

Spracovanie prebieha nasledovne. Súbor domény alebo problému sa prečíta znak po znaku, z písmen sa poskladajú slová a zo slov zoznam slov (program `readFile.pl`). Následne je zoznam spracovaný DCG gramatikou, ktorá vráti prologovský term (program `parseDomain.pl` a `parseProblem.pl`).

V tabuľke 4.1 je príklad vstupu a výstup z parséra. Parsér zachováva poradie parametrov a celkovú logickú štruktúru PDDL súboru. Čo je výhodou pre ďalšie spracovanie v Prologu.

## 4.2 Plánovače

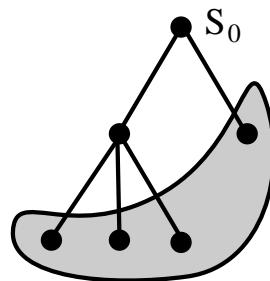
V tejto časti bližšie popíšeme implementáciu algoritmov z teoretickej časti. Zamieríame sa na odstránenie nedeterminizmu pomocou rôznych spôsobov prehľadávania. Aby sme ešte urýchliли nájdenie plánu doplníme plánovače ešte o heuristiku.

### 4.2.1 Spôsob prehľadávania

Na úvod si pripomeňme spoločné črty a rozdiely dopredného a spätného (liftovaného) plánovania. Všetky tri algoritmy nedeterministicky prehľadávajú priestor. Pričom dopredné prehľadávanie prehľadáva stavový priestor, spätné prehľadáva priestor podielov a liftovaná verzia spätného prehľadáva čiastočne

Súbor PDDL	Term v Prologu
<pre> ... (:action pick-up :parameters (?x - block) :precondition (and (clear ?x)   (ontable ?x) (handempty)) :effect (and (not (ontable ?x))   (not (clear ?x))   (not (handempty))   (holding ?x)) ) ... </pre>	<pre> ... action(pick-up, [(block[?x])],  [clear(?x),   ontable(?x), handempty], [holding(?x)], //positiv [ontable(?x), //negativ   clear(?x), //effects   handempty], []) ... </pre>

Tabuľka 4.1: Ukážka spracovania PDDL súboru do prologovského termu.



Obr. 4.1: Šedou farbou je označený okraj prehľadávania

inštanciovaný stavový priestor. Všetky tri algoritmy využívajú na výber d'alsieho kroku nedeterminizmus. Pri praktickej implementácii je potrebné nahrať nedeterminizmus prehľadávaním.

V nasledujúcich odstavcoch uvedieme základné spôsoby prehľadávania, ktoré môžu byť použité vo všetkých troch prehľadávaniach. Pre jednoduchosť sa zameriame iba na dopredné prehľadávanie. Spätné prehľadávanie sa robí analogicky.

Prehľadávacie algoritmy, ktoré popíšeme, sú založené na udržovaní okraju. Okraj je množina stavov, ktoré čakajú na preskúmanie. Na začiatku obsahuje okraj iba inicializačný stav  $s_0$ . Postupne expandujeme stavy z okraju až kým nenájdeme cieľový stav. Pod expandovaním stavu  $s$  rozumieme zmazanie stavu  $s$  z okraja a pridanie  $N_s$  do okraju, kde  $N_s = \{s' | \exists a \in A : s' = \gamma(s, a)\}$ .

Aký vrchol sa vyberie z okraja určuje spôsob prehľadávania.

### Prehľadávanie do hĺbky (DFS)

Pri prehľadávaní do hĺbky sa expanduje vždy naposledy pridaný vrchol. Najčastejšie sa implementuje rekurzívne. Pri tomto riešení algoritmus aplikuje možnú akciu na aktuálny stav a presunie sa do nového stavu. Ak na stav nie je

aplikovateľná žiadna akcia, tak sa program vráti k predošlému stavu a skúsi pokračovať ďalšou akciou do nového stavu.

Aby algoritmus dokázal prejsť celý stavový priestor a tým garantoval nájdenie riešenia, je nutné zabrániť opakovanému prechádzaniu rovnakých stavov. Algoritmus si udržuje zoznam všetkých stavov v aktuálnej ceste. Ak sa nový stav už nachádza v zozname, tak došlo k zacykleniu.

Pri hľadaní optimálneho plánu program neskončí pri nájdení prvého plánu. Naopak program pokračuje v prehľadávaní celého stavového priestoru. To však nie je nutné a stačí sa obmedziť na iba na cesty, ktoré nie sú dlhšie ako doteraz najlepší nájdený plán.

Heuristika predstavuje vhodné rozšírenie programu. Určí v akom poradí sa budú aplikovať akcie na aktuálny vrchol a nasmeruje prehľadávanie k cielu, čím sa zmenší veľkosť prehľadávaneho priestoru.

Prehľadávanie do hĺbky má malé pamäťové nároky - stačí si pamätať vrcholy v aktuálnej ceste. Na druhú stranu v prípade nepresnej alebo žiadnej heuristiky je často nutné prehľadať cely stavový priestor.

## Prehľadávanie do šírky (BFS)

Pri prehľadávaní do šírky expandujeme vždy najneskôr pridaný vrchol. Tým sa zabezpečí, že sa stavový priestor prehľadáva postupne podľa vzdialnosti od inicializačného vrcholu. V praxi sa implementuje pomocou fronty. Pri prehľadávaní vrcholu sa určia všetky nasledujúce vrcholy a tie sa uložia na koniec fronty. Následne sa pokračuje vrcholom zo začiatku fronty.

Tak ako v prehľadávaní do hĺbky je vhodné predísť zacykleniu. Preto je nutné si uchovať zoznam všetkých navštívených vrcholov. Pred spracovaním ďalšieho vrcholu je nutné skontrolovať, či už raz neboli navštívený a ak áno, tak tento stav vynechať.

Vzhľadom na to, že sa stavový priestor prehľadáva po vrstvách, tak poradie v ktorom sa pridávajú vrcholy do fronty nie je dôležité. V najhoršom prípade bude cieľ objavený ako posledný vrchol v aktuálne prehľadávanej vrstve. Preto heuristika v tejto metóde nepomôže.

Ked'že je potrebné si pamätať všetky už navštívené vrcholy, má prehľadávanie do šírky veľké pamäťové nároky. Čo predstavuje podstrom stavového priestoru až do hĺbky, ktorá je rovná dĺžke optimálneho plánu.

## Iteratívne prehľadávanie do hĺbky (IDDFS)

Jedná sa o rozšírenie prehľadávania do hĺbky. Začína tým, že sa spustí prehľadávanie do hĺbky s maximálnou hĺbkou  $D = 1$ . Ak neboli nájdený plán, tak sa spustí prehľadávanie do hĺbky s maximálnou hĺbkou  $D = 2$ . Ak opäť neboli plán nájdený, tak sa opäť zvyšuje maximálna hĺbka o jedna až kým nie je nájdený plán. Ak neboli navštívený žiadnený vrchol v hĺbke  $D$ , tak sme prehľadali celý stavový priestor a algoritmus končí.

Pri tomto prehľadávaní heuristika nepomôže a to z rovnakého dôvodu ako bolo uvedené v prehľadávaní do šírky.

Iteratívne prehľadávanie do hĺbky má rovnaké pamäťové nároky ako prehľadávanie do hĺbky. Nevýhodou je, že sa opakovane prehľadávajú už prehľadané stavy a tým sa zvyšuje časová náročnosť.

### A\*

Pri tomto prehľadávaní vyberáme z okraja stav, ktorý je najbližšie k cieľovému stavu. K ohodnoteniu stavu  $s$  sa používa heuristická funkcia  $h(s, g)$ , ktorá odhaduje počet akcií nutných k dosiahnutiu cieľa  $g$ . Spolu s aktuálnou hĺbkou stavu  $g(s)$  tvoria celkový odhad pre vzdialenosť inicializačného stavu k cieľu.

Okraj sa implementuje pomocou prioritnej fronty, ktorá zabezpečuje, že sa ako vyberie stav s najnižším ohodnotením a teda pravdepodobne najbližšie k cieľu.

Zacykleniu je najjednoduchšie zabrániť, tak ako pri prehľadávaní do šírky. Program si udržuje zoznam už navštívených vrcholov, čo mu zodpovedá pamäťová náročnosť.

Časová náročnosť závisí od presnosti odhadu heuristickej funkcie. V praxi je dôležité nájsť rovnováhu medzi presnosťou odhadu a jednoduchosťou výpočtu heuristickej funkcie. Viaceré heuristiky budú popísane v časti 4.2.2

### WA\*

Malou úpravou prehľadávacieho algoritmu A\* je možné získať algoritmus WA\*. Jediný rozdiel spočíva vo vynásobení heuristickej funkcie konštantou  $W \geq 1$ . Pre  $W = 1$  sa jedná opäť o algoritmus A\*. Význam zavedenia konštanty bude vysvetlený v príklade 4.2.2.

## 4.2.2 Heuristika

V predošej časti sme popísali implementáciu prehľadávacích algoritmov, z ktorých väčšina používala heuristickú funkciu. V tejto časti zavedieme presne pojemy heristika a popíšeme tri základné heuristiky pre prehľadávanie v plánovaní.

**Definícia 4.2.1.** *Heuristicou funkciou  $h(s, g)$  nazývame funkciu, ktorá pre daný stav  $s$ , cieľ  $g$  a plánovaciu doménu odhadne vzdialenosť stavu  $s$  od cieľa  $g$  v stavovom priestore domény.  $h^*(s, g)$  označujeme najmenšiu vzdialenosť medzi  $s$  a  $g$ .*

**Definícia 4.2.2.** *Ak pre všetky stavy  $s$  platí, že  $h(s, g) \leq h^*(s, g)$ , tak hovoríme, že heuristická funkcia je prípustná. Prehľadávací algoritmus A\* s prípustnou funkciou nájde vždy optimálny plán.*

**Príklad 4.2.1.** Zoberme si ako príklad dopredné prehľadávanie, kde stav je reprezentovaný ako množina inštanciovaných predikátov. Heuristickú funkciu definujeme nasledovne  $h(s, g) = |g - s|$  tzn. ako počet chýbajúcich predikátov v stave  $s$  k splneniu cieľa  $g$ . Je jasné, že heuristická funkcia nie je prípustná. Nech k splneniu cieľa sú potrebné ešte 3 predikáty a nech existuje akcia, ktorá je aplikovateľná na stav  $s$  a zároveň pridáva chýbajúce predikáty. Potom  $h(s, g) = 3$ , ale  $h^*(s, g) = 1$ .

**Príklad 4.2.2.** Vynásobením prípustnej heuristickej funkcie ľubovoľnou konštantou  $W > 1$  získame novú heuristickú funkciu, ktorá však už nie nutne je prípustná. Z tohto dôvodu algoritmus  $WA^*$  nemusí nájsť optimálny plán. Na druhú stranu vyššia hodnota  $W$  nasmeruje algoritmus rýchlejšie k cieľu.

### Heuristika $h_0$

Jedná sa o veľmi optimistickú heuristiku, kde  $h(s, g) = 0$ . Má za účel ukázať ako by sa plánovače správali bez heuristiky. Zároveň slúži na absolútne porovnanie heuristik.

### Heuristika $h_{\text{diff}}$

Je jednoduchá heuristika, ktorá bola uvedená v príklade 4.2.2, definovaná ako  $h(s, g) = |g - s|$ .

### Heuristika $h_{\text{add}}$

Táto heuristika bola prevzatá z [11]. Heuristika  $h_{\text{add}}$  sa snaží zjednodušiť plánovací problém a to tým že neuvažuje negatívne efekty akcií. Takúto akciu nazývame relaxovaná akcia. Určenie heuristiky prebieha nasledovne. Pozrieme sa na jednotlivé predikáty v ciele  $g$  a zistíme kolko relaxovaných akcií je potrebné na splnenie tohto predikátu z aktuálneho stavu. Súčet počtu potrebných akcií pre všetky predikáty v ciele udáva heuristiku. Formálne je to možné vyjadriť nasledovne:

$$h_{\text{add}}(s, g) = G_s(g)$$

$$G_s(C) = \sum_{p \in C} g_s(p)$$

$$g_s(p) = \begin{cases} 0 & \text{if } p \in s \\ \min_{a \in A \wedge p \in \text{effect}^+(a)} [1 + G_s(\text{precond}(a))] & \text{inak} \end{cases}$$

Niekterá akcia však môže splniť viacero predikátov z cieľa a pri tom je zarátaná viackrát. Je preto vidieť, že táto heuristika je neprístupná.

### Heuristika $h_{\text{max}}$

Heuristika  $h_{\text{max}}$  sa lísi od predchádzajúcej len v definícii funkcie  $G_s(C)$ . Namiesto sumy sa tu uvažuje len maximum:

$$h_{\text{max}}(s, g) = G_s(g)$$

$$G_s(C) = \max_{p \in C} g_s(p)$$

Tento zmenou v definícii sme definovali prípustnú heuristiku. Avšak funkcia maximum stratila cenu jednotlivých predikátov a tým stratila informačnú hodnotu.

Pri doprednom prehľadávaní je cieľ  $g$  zafixovaný a mení sa stav  $s$ . Je dôležité si uvedomiť, že pri spätnom prehľadávaní je to presne naopak. Čo sa dá využiť a predpočítať si funkciu  $g_s(p)$  pre všetky predikáty pri inicializácii.

Smer prehľadávania	Spôsob prehľadávania	Heuristika
<ul style="list-style-type: none"> <li>• Dopredné</li> <li>• Spätné           <ul style="list-style-type: none"> <li>– liftované</li> </ul> </li> </ul>	<ul style="list-style-type: none"> <li>• DFS</li> <li>• BFS</li> <li>• IDDFS</li> <li>• A*</li> <li>• WA*</li> </ul>	<ul style="list-style-type: none"> <li>• <math>h_0</math></li> <li>• <math>h_{\text{diff}}</math></li> <li>• <math>h_{\text{add}}</math></li> <li>• <math>h_{\text{max}}</math></li> </ul>

Tabuľka 4.2: Prehľad univerzálnych modulov plánovača. Ľubovolným skombinovaním smeru prehľadávania, spôsobu prehľadávania a heuristiky získame plánovač.

### 4.2.3 Moduly

V predošlých častiach sme dekomponovali plánovač do troch časti: smer prehľadávania, spôsob prehľadávania a heuristika (tabuľka 4.2). Toto rozdelenie je nielen vhodné pre jednoduchší rozbor implementácie, ale presne zodpovedá rozdeleniu plánovača do modulov. V module je implementovaná konkrétny smer, spôsob alebo heuristika. Samotný plánovač sa potom bude skladať z ľubovolnej zmyslupnej kombinácie troch modulov.

**Príklad 4.2.3.** Zoberme si dopredný plánovač s prehľadávaním do hĺbky a heuristikou  $h_{\text{diff}}$ . Už samotný názov napovedá z akých modulov sa bude plánovač skladať. Nie všetky kombinácie však dávajú zmysel. Napríklad prehľadávanie do šírky nemá zmysel kombinovať s heuristikou.

Takéto riešenie je prehľadné a škálovateľné. Pri pridávaní napr. ďalšej heuristiky nie je potrebné zasahovať do už existujúcich modulov.



# Kapitola 5

## Porovnanie

V tejto kapitole sa budeme venovať porovnaniu implementovaných prehľadávacích techník na reálnych problémoch. V úvode popíšeme výber testovacích domén a metodiku testovania. V nasledujúcej časti vyhodnotíme ako sa správali rôzne plánovače na jednotlivých doménach.

### Metodika testovania

Aby sa zabezpečila relevantnosť výsledkov, tak k testovaniu boli použité plánovacie problémy z medzinárodnej plánovacej súťaže IPC [1]. Stručný popis domén je uvedený v prílohe A.

K testovaniu bol použitý počítač s operačným systémom Debian a SICStus Prolog. Jednotlivé plánovače boli spustené na dole uvedenej počítačovej zostave po dobu 500 sekúnd.

- Procesor: Intel® Celeron® CPU 2.4 GHz
- Pamäť: 1 GB
- Operačný systém: Debian 2.6.26-2-686
- Prolog: SICStus 4.0.8 (x86-linux-glibc2.7)

Celkový čas behu plánovača a spotreba pamäti bola získavaná vstavaným predikátom **statistics/2**. Okrem toho nás zaujímala dĺžka plánu a počet navštívených vrcholov pri prehľadávaní.

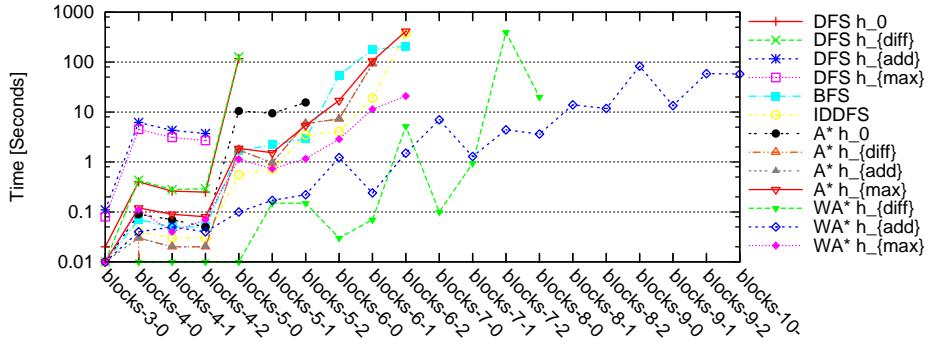
Pre prehľadávací algoritmus WA\* bola konštantá  $W$  zafixovaná na hodnote 5. Iné hodnoty neboli bližšie skúmané.

V prílohe C sú výsledky spracované v podobe grafov.

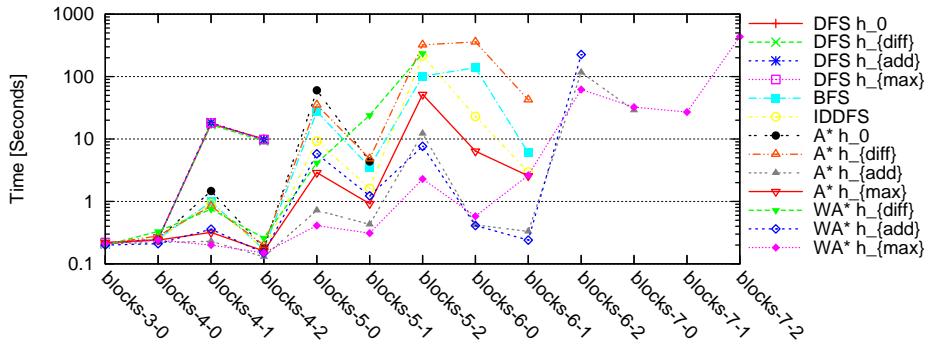
### 5.1 Blocks World

Najjednoduchší problém pre túto doménu pozostáva z postavenia veže z troch kociek. Obťaženosť sa stupňuje s postupným pridávaním kociek.

Graf 5.1: Doba výpočtu dopredného prehľadávania pre doménu Blocks World



Graf 5.2: Doba výpočtu spätného prehľadávania pre doménu Blocks World



## Dopredné prehľadávanie

Najhoršie v tejto doméne dopadlo prehľadávanie do hĺbky. Použitá heuristika pri tom len zhoršila bežiaci čas. Druhou skupinou tvoria algoritmy, ktoré prehľadávajú priestor postupne: BFS, IDDFS, WA\*  $h_{\max}$  a variácie A\*. Napriek tomu, že iteratívne prehľadávanie musí navštíviť stavy opakovane, tak rýchlejšie našlo skoro pre všetky problémy.

Zaujímave je, že celkom jednoduchá heuristika  $h_{\text{diff}}$  funguje celkom dobre a rýchlo. Vo svete kociek je dôležité v akom poradí sú ukladané kocky. Napriek tomu táto heuristika dopadla veľmi dobre.

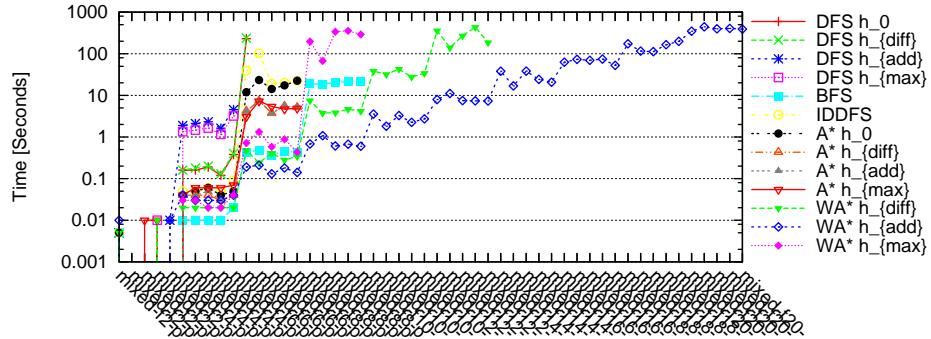
Najviac problémov vyriešil algoritmus WA\* s heuristikou  $h_{\text{add}}$ . Dĺžky nájdenej plánov sú približne dva krát dlhé ako optimálny plán.

## Spätné prehľadávanie

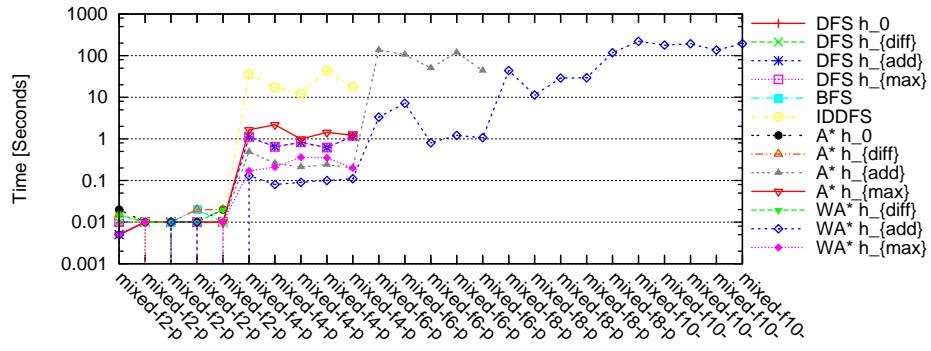
V spätnom prehľadávaní dominuje prehľadávanie IDDFS. Napriek tomu je rozdiel veľmi malý. Tesným rivalom je BFS. Avšak pri dlhších plánoch sa prejavila rýchlosť iteratívneho prehľadávania do hĺbky. Celkovo plánovač WA\*  $h_{\max}$  vyriešil najviac problémov a pritom s optimálnym plánom.

Hlavným problémom spätného prehľadávania je vysoký vetviaci faktor, ktorý je spôsobený zakázanými stavmi. V tejto doméne bolo nájdených len zopár

Graf 5.3: Doba výpočtu dopredného prehľadávania pre doménu Elevators



Graf 5.4: Doba výpočtu spätného prehľadávania pre doménu Elevators



zakázaných dvojíc, takže detektia prebieha len obmedzene.

## 5.2 Elevators

V doméne Elevators je potrebné naplánovať prepravu osôb výtahom. Obťažnosť problému stúpa s počtom osôb, ktoré je potrebné prepraviť.

### Dopredné prehľadávanie

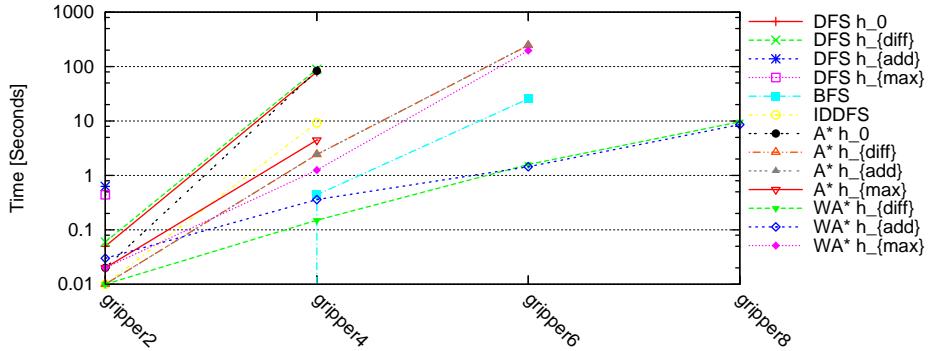
Tak ako v doméne Blocks World, prehľadávanie do hĺbky nie je pre túto doménu vhodné. DFS prehľadáva stavový priestor niekde hlboko, čo je vidieť z veľkého počtu navštívených vrcholov. Druhým najhorším prehľadávaním je IDDFS, ktoré tiež prehľadávalo privela stavov. Z čoho sa dá usúdiť, že doména Elevators má veľký vetviaci faktor už v malých hĺbkach.

BFS obstalo najlepšie z pomedzi optimálnych plánovačov. Viac problémov už vyriešil iba WA\*  $h_{add}$ , avšak s trošku dlhšími plánmi.

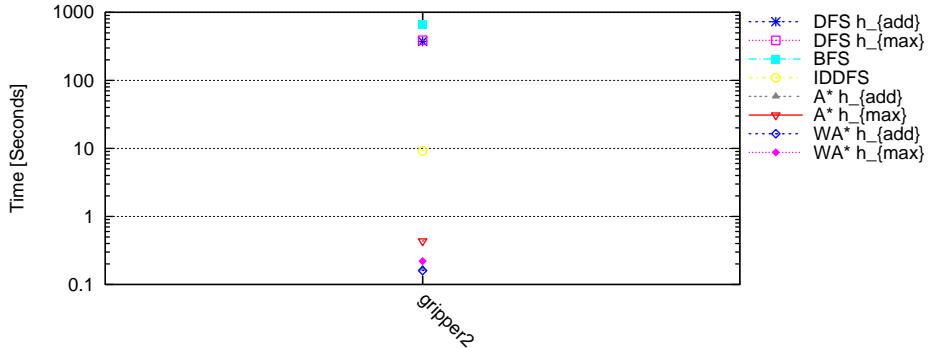
### Spätné prehľadávanie

Tak ako pri doprednom plánovaní heuristika  $h_{add}$  najlepšie odhadovala vzdialenosť k cieľu. Celkovo je táto doména pre spätné prehľadávanie príliš zložitá.

Graf 5.5: Doba výpočtu dopredného prehľadávania pre doménu Gripper



Graf 5.6: Doba výpočtu spätného prehľadávania pre doménu Gripper



## 5.3 Gripper

Pre túto doménu boli použité štyri jednoduché problémy. Robot musí preniesť lôpy z jednej miestnosti do druhej. Pričom náročnosť úlohy sa stupňuje s počtom lôpt od 2 do 8.

### Dopredné prehľadávanie

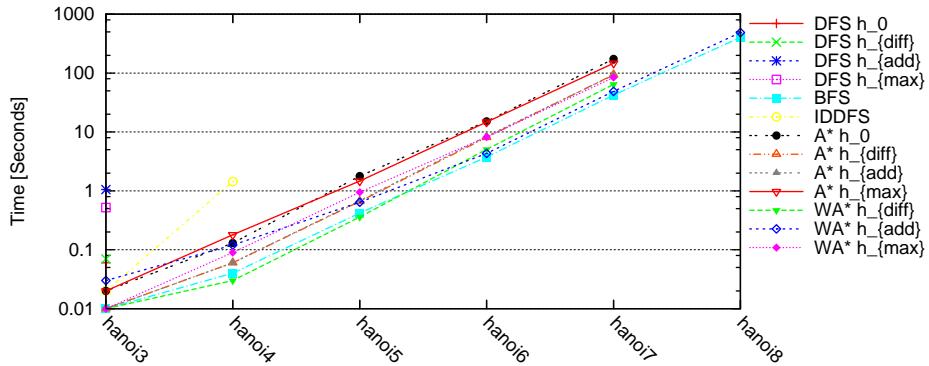
Kedže lôpy sa presúvajú nezávisle na seba, nie je dôležité v akom poradí sa presunú. Napriek tomu musia plánovače vyskúšať všetky kombinácie akcií. Najmenej problémom vyriešili plánovače postavené na prehľadávaní do hĺbky. S desaťnásobne kratším časom, ale s rovnakým počtom úloh skončil aj plánovač IDDFS.

Predel' medzi heuristikami  $A^*$  a  $WA^*$  určuje BFS. Zaujímavé je, že heuristiky ešte naviac oddialili nájdenie cieľa. Jedine heuristiky pre  $WA^* h_{\text{max}}$  a  $WA^* h_{\text{add}}$  vyriešili všetky problémy, ale so zbytočne dlhým plánom.

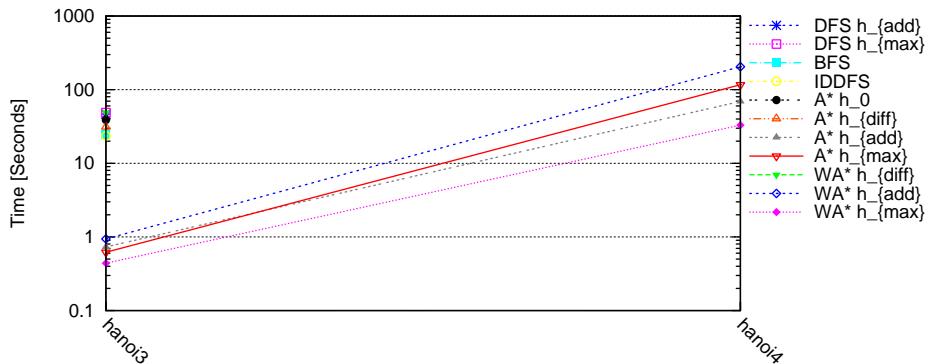
### Spätné prehľadávanie

Veľmi stabilné sa ukázalo prehľadávanie do šírky, ktoré vyriešilo najviac úloh. Avšak pri plánovacích problémoch s dlhším plánom sa ukázalo byť IDDFS

Graf 5.7: Doba výpočtu dopredného prehľadávania pre doménu Hanoi



Graf 5.8: Doba výpočtu spätného prehľadávania pre doménu Hanoi



rýchlejšie.

## 5.4 Hanoi

V tejto doméne je úlohou plánovača presunúť kotúče z prvého kolíka na tretí. Číslovanie plánovacích problémov zodpovedá počtu kotúčov, ktoré je potrebné presunúť.

### Dopredné prehľadávanie

Aj v tejto doméne prehľadávanie do hlíbky nebolo úspešné a vyriešilo iba jeden problém. IDDFS sa nedokázalo vysporiadať s veľkým vetviacim faktorom a vyriešilo dva problémy. Zvyšné algoritmy založené na heuristikách našli plány rýchlejšie, ale zaroveň aj podstatne dlhšie. BFS pritom v podobnom čase vyriešilo o jeden problém viac. Celkovo sa dá povedať, že použité heuristiky boli nevhodné pre túto doménu.

### Spätné prehľadávanie

Pri spätnom prehľadávaní sa prehľadávajú nezmyselné stavov, ktoré vedú do ďalších nezmyselných stavov. Takto sa neskutočne zväčšíl stavový priestor tak

jednoduchej hry. Plánovače vyriešili iba prvý problém s tromi kotúčmi. Plánovač BFS musel pri spätnom prehľadávaní prezrieť 1550 podielov, ale rovnaký plánovač pri doprednom prehľadávaní prezrel iba 25 stavov.

# Kapitola 6

## Záver

Dopredné a spätné plánovanie sú najjednoduchšie klasické plánovacie techniky. Predstavujú základ pre štúdium plánovania. V tejto práci sme implementovali spomenuté techniky formou piatich rôznych metód prehľadávania a tieto metódy sme d'alej rozšírili o štyri heuristiky. Získali sme tak pestrú vzorku plánovačov vhodnú na porovnanie.

Aby bolo zaručená objektivita výsledkov, porovnanie prebehlo na doménach, ktoré boli použité na medzinárodnej súťaži v plánovaní.

Nedá sa jednoznačne určiť najlepší plánovač. Výsledky sú vždy pevne späte s testovanou doménou. Však plánovače postavené na prehľadávaní do hĺbky dopadli v testoch najhoršie a nie sú vhodné na plánovanie. Z optimálnych dopredných plánovačov sú najlepšie BFS a IDDFS, z ktorých aspoň jeden vždy vynikol vo všetkých doménach. Veľa problémov vyriešil dopredný plánovač WA\*  $h_{\text{diff}}$ . Disponuje pritom len veľmi jednoduchou a ľahko vypočítateľnou heuristikou. O to viac prekvapilo, že jeho plány boli len niekoľko akcií dlhšie ako optimálny plán. Na testovaných doménach si dobre počína aj plánovač WA\*  $h_{\text{add}}$ , ktorého plány však boli nezriedka dvojnásobne dlhé. Heuristika  $h_{\max}$  obsahuje v porovnaní s  $h_{\text{add}}$  viac informácií pre prehľadávaní. Napriek tomu nevynikla ani v jednej doméne.

Plánovače založené na spätnom prehľadávaní dopadli celkovo horšie. Hlavnou príčinou sú zakázané stavby, ktoré niekoľkonásobne zväčšujú stavový priestor. Tu zostáva ešte priestor pre d'alej štúdium a návrh lepšej detekcie zakázanych stavov. Ďalším zrýchlenie by sa mohlo dosiahnuť implementovaním liftovanej verzie spätného plánovania, ktoré bolo vysvetlené v teoretickej časti práce.

Súčasťou práce je aj parser plánovacieho jazyka PDDL napísaný v Prologu, ktorý môže nájsť uplatnenie aj v iných plánovačoch založených na programovačom jazyku Prolog.



# Literatúra

- [1] 2nd international planning competition.  
<http://www.cs.toronto.edu/aips2000/>.
- [2] 5th international planning competition.  
<http://zeus.ing.unibs.it/ipc-5/>.
- [3] Artificial intelligence planning systems.  
<http://www.cs.cmu.edu/~aips98/>.
- [4] Graphical interface for planning with objects.  
<http://compeng.hud.ac.uk/planform/gipo/>.
- [5] itsimple (integrated tools software interface for modeling planning environments).  
<http://dlab.poli.usp.br/twiki/bin/view/ItSIMPLE/WebHome>.
- [6] *LISP*.  
[http://en.wikipedia.org/wiki/Lisp\\_\(programming\\_language\)](http://en.wikipedia.org/wiki/Lisp_(programming_language)).
- [7] Writing planning domains and problems in pdll.  
<http://users.rsise.anu.edu.au/~patrik/pddlman/writing.html>.
- [8] *Definite clause grammar*, wikipedia, the free encyclopedia, 2009.  
<http://en.wikipedia.org/wiki/DCG>.
- [9] *Prolog* wikipedia, the free encyclopedia, 2009.  
<http://en.wikipedia.org/wiki/Prolog>.
- [10] A.L. Blum and M.L. Furst. Fast planning through planning graph analysis. *Artificial intelligence*, 90(1-2):281–300, 1997.
- [11] B. Bonet and H. Geffner. Planning as heuristic search. *Artificial Intelligence*, 129(1-2):5–33, 2001.
- [12] Alfonso Gerevini and Derek Long. Bnf description of pddl3.0.  
<http://www.cs.yale.edu/homes/dvm/papers/pddl-bnf.pdf>, 2005.
- [13] M. Ghallab, D.S. Nau, and P. Traverso. *Automated Planning: theory and practice*. Morgan Kaufmann Publishers, 2004.
- [14] J. Hoffmann. FF: The fast-forward planning system. *AI magazine*, 22(3):57, 2001.

- [15] N.J. Nilsson. *Principles of artificial intelligence*. Birkhäuser, 1982.
- [16] P. Töpfer. Algoritmy a programovací techniky. *Prometeus, Praha*, 1995.

# Dodatok A

## Domény

### Blocks World

Svet kociek vznikol ako ukážkový program umelej inteligencie, ktorý vytvoril Terry Winograd. Následne sa stal najznámejšou plánovacou doménou.

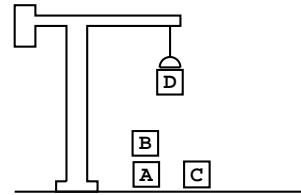
Svet pozostáva z rovnej plochy - stola. Na stole sú po rozkladané kocky, ktoré sú pre jednoduchosť označené písmenami. Kocka môže ležať na stole alebo na práve jednej inej kocke. Na stole je tiež robot s jednou robotickou rukou, ktorý môže hýbať s kockami. Robot dokáže držať maximálne jednu kocku. Pre zdelenie nie je dôležitá absolútna poloha kocky tzn. že popis nerozlišuje medzi kockou umiestnenou na ľavom okraji stola alebo na pravom okraji stola.

Robot dokáže vykonať nasledujúce úkony:

- `stack(A,B)` polož kocku A na voľnú kocku B.
- `unstack(A,B)` zober voľnú kocku A z kocky B.
- `pick-up(A)` zdvihni voľnú kocku A zo stola.
- `put-down(A)` polož voľnú kocku A na stôl.

Ľubovoľnú situáciu na stole je možné zapísat' nasledujúcimi predikátmi. Pričom je použitá klasická reprezentácia stavu, ktorá je bližšie popísaná v časti 2.2.

- `ontable(A)` kocka A leží na stole.
- `clear(A)` na kocke A nie je iná kocka, inými slovami kocka je voľná.
- `holding(A)` ruka robota drží kocku A.
- `handempty` ruka robota nedrží žiadnu kocku.
- `on(A,B)` kocka A leží na kocke B.



Obr. A.1: Príklad zo sveta kociek

**Príklad A.0.1.** Situáciu na obrázku A.1 popisuje stav: `ontable(A)`, `on(B,A)`, `clear(B)`, `holding(D)`, `ontable(C)`, `clear(C)`

V dodatku B je doména Blocks World použitá ako ukážkový príklad zápisu domény v jazyku PDDL.

## Elevators

Ako už názov napovedá, jedná sa o naplánovanie chodu výtahu. Doména pozostáva z jedného výtahu s neobmedzenou kapacitou, ktorý sa pohybuje medzi poschodiami. Ďalej sú tu viacerí pasažieri, ktorí sa chcú dostať z jedného poschodia na iné. Stav je popísaný ako pozícia výtahu, usporiadanie poschodí a zoznam pasažierov s ich počiatočným a konečným poschodím. Plán pozostáva z akcií

- `board(floor3, passenger1)` - 1. pasažier nastúpi na 3.poschodí.
- `depart(floor4, passenger1)` - 1. pasažier vystúpi na 4.poschodí.
- `up(floor1, floor6)` - presun výtahu hore z prvého poschodia na šieste poschodie.
- `down(floor3, floor2)` - presun výtahu dole z tretieho poschodia na druhé poschodie.

Je dôležité si uvedomiť, že optimálny plán minimalizuje počet zastavení výtahu a nie počet prejdených poschodí ako by sa mohlo zdať.

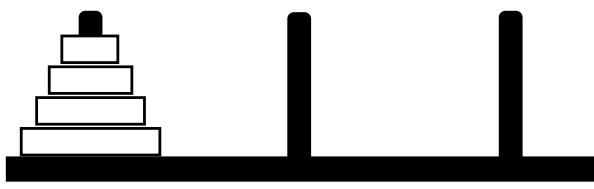
## Gripper

Táto doména simuluje robotický svet. Majme jedného robota s dvomi rukami, do ktorých môže nezávisle uchopíť loptu. Robot sa pohybuje medzi miestnosťami, v ktorých sa nachádzajú lopy. Pričom všetky miestnosti sú navzájom spôsobené. Cieľom robota je popresúvať lopy podľa zadania.

## Hanoi

Hanojské veže je matematický hlavolam. Na začiatku sú kotúče rôznej veľkosti nasadené na jednom kolíku z troch kolíkov (obrázok A.2). Kotúče sú usporiadane od najmenšieho po najväčší. Úlohou je presunúť všetky kotúče na iný kolík. Pričom v jednom ťahu je možné premiestniť práve jeden kotúč a vždy presúvame vrchný kotúč z jedného kolíka na iný kolík s kotúčom väčšieho polomeru.

Úlohou plánovača je vyriešiť hru hanojské veže tzn. presunúť celý stĺpec kotúčov z jedného kolíku na druhý. Obtiažnosť úlohy stúpa s počtom kotúčov.



Obr. A.2: Začiatočná pozícia štyroch kotúčov v hre hanojské veže.



## Dodatok B

# Planning Domain Definition Language (PDDL)

V tomto dodatku je popísaná základná štruktúra doménového a problémového PDDL súboru, základné rozšírenia a stručný vývoj jazyka od jeho vzniku. Presnú štruktúru je možné nájsť v BNF popise jazyka PDDL[12].

PDDL je jazyk na popis plánovacieho problému. Samotná syntax PDDL je odvodená od syntaxi programovacieho jazyka LISP[6]. Skladá sa z definície domény a definície problému. *Napriek tomu, že to špecifikácia nevyžaduje, niektoré plánovače vyžadujú, aby boli definície v osobitných súboroch*[7].

Na úpravu postačí ľubovoľný textový editor, avšak pri písaní väčšej domény s reálnymi dátami je vhodné použiť aplikáciu s grafickým rozhraním napr. GIPO (Graphical Interface for Planning with Objects)[4] alebo itSIMPLE (Integrated Tools Software Interface for Modeling Planning Environment)[5].

### B.1 Definícia domény

Doménový súbor pozostáva z povinných a nepovinných klauzúl.

V klauzule **predicates** je pre plánovač uvedený zoznam predikátov, ktoré zodpovedajú klasickej reprezentácii spomenutej v kapitole prehľad klasických techník. Volné premenné sú označené prefixovým otáznikom, napr. **?x**.

Ako posledné sú uvedené dostupné akcie v doméne. Každá akcia je špecifikovaná ako zoznam parametrov, predpokladov a efektov. Negatívne efekty sú označené prefixom **not**.

Doména ďalej špecifikuje minimálne schopnosti plánovača (Requirements flags). Tu spomeniem dve rozšírenia, ktoré sú vyžadované skoro každou doménu.

Rozšírenie **:typing** umožňuje rozdelenie objektov do typov. Zoznam všetkých dostupných typov v doméne je uvedený v klauzule **types**. Objekt alebo premenná môže byť označený za pomlčkou typom. Viaceré typy môžu byť zlúčené do skupín.

Štandardne sa za riešenie plánovacieho problému považuje plán určený minimálnym počtom akcií. Rozšírenie **:action-cost** umožňuje zaviesť ľubovoľnú metriku. Akcia môže mať ako svoj efekt uvedenú cenu. V klauzule

:**metrics** v definícii problému je určená funkcia, ktorú má plánovač minimizovať alebo maximalizovať.

V časti História sú spomenuté ďalšie schopnosti a ich historický prehľad.

Príklad 1: Definícia domény pre svet kociek.

```
(define (domain BLOCKS-WORLD)
  (:requirements :strips :typing :action-costs)
  (:types block)
  (:predicates (on ?x - block ?y - block)
               (ontable ?x - block)
               (clear ?x - block)
               (handempty)
               (holding ?x - block)
               )
  (:functions (total-cost) - number)
  ;; List of actions
  (:action pick-up
    :parameters (?x - block)
    :precondition (and (ontable ?x) (clear ?x) (handempty))
    :effect (and (not (ontable ?x)) (not (clear ?x))
                  (not (handempty)) (holding ?x)
                  (increase (total-cost) 2)))
  (:action put-down
    :parameters (?x - block)
    :precondition (holding ?x)
    :effect (and (not (holding ?x))
                  (clear ?x)
                  (handempty)
                  (ontable ?x)
                  (increase (total-cost) 2)))
  (:action stack
    :parameters (?x - block ?y - block)
    :precondition (and (holding ?x) (clear ?y))
    :effect (and (not (holding ?x))
                  (not (clear ?y))
                  (clear ?x)
                  (handempty)
                  (on ?x ?y)
                  (increase (total-cost) 2)))
  (:action unstack
    :parameters (?x - block ?y - block)
    :precondition (and (on ?x ?y) (clear ?x) (handempty))
    :effect (and (holding ?x)
                  (clear ?y)
                  (not (clear ?x))
                  (not (handempty))
                  (not (on ?x ?y))
                  (increase (total-cost) 2)))
  )
)
```

V príklade 1 je výpis súboru v jazyku PDDL, ktorý popisuje jednoduchý svet kociek (**Blocks World**). Jediným objektom v tejto doméne sú kocky.

Kocka môže ležať na stole, byť položená na inej kocke alebo v ruke. K opísaniu polohy kocky slúžia predikáty: (`ontable A`), (`on B C`), (`holding D`). Predikát (`clear A`) značí, že na kocke A nie je žiadna iná kocka. Bez parametrových predikátov (`handempty`) značí, že nedrží žiadnu kocku. V našom jednoduchom svete je iba jedna ruka, preto je jasné ktorú máme na mysli. Práve spomenuté predikáty úplne popisujú situáciu na stole, za predpokladu, že nás nezaujíma, kde presne sa nachádzajú kocky na stole.

Svet kociek je dynamický a preto doména obsahuje zoznam akcií: zdvihni kocku zo stola, polož kocku na stôl, zdvihni kocku z inej kocky a polož kocku na inú kocku. Každá akcia pozostáva z predpokladov a efektov.

Napríklad pre akciu zdvihni kocku A zo stola musia byť splnené nasledujúce predpoklady. Kocka A leží na stole, na kocke neleží žiadna iná kocka a ruka je prázdna. Aplikovaním akcie sa pridá do aktuálneho stavu predikát (`holding A`) a odoberú predikáty (`ontable A`), (`clear A`), (`handempty`). Nakoniec sa zvýsi hodnota funkcie `total-cost` o dva.

Podobne sú popísané aj zvyšné akcie. Je dôležité si všimnúť, že jazyk PDDL presne zodpovedá klasickej reprezentácií.

## B.2 Definícia problému

Opäť je definícia tvorená viacerými klauzulami. Prvé dve špecifikujú názov problému a meno domény, pre ktorú je problém určený. V nasledujúcej klauzule `objects` sú uvedené všetky objekty a ich typ.

Inicializačný stav je reprezentovaný ako zoznam inštancií predikátov v časti `init`. Cieľ je uvedený ako formula zložená z operácie `and` a inštancií predikátov. V prípade ďalších rozšírení môže formula obsahovať negácie, implikácie a kvantifikačné operátory.

Rozšírenie `action-cost` bolo už spomenuté v predošej časti. Slúži na zavedenie ľubovoľnej metriky. Plánovač v tom prípade nehľadá plán s najmenším počtom akcií, ale minimalizuje alebo maximalizuje funkciu uvedenú v časti `metric`.

Príklad 2: Príklad definície problému pre svet kociek.

```
(define (problem BLOCKS-WORLD-1)
  (:domain BLOCKS-WORLD)
  (:objects D B A C - block)
  (:init (clear C) (clear A)(clear B) (clear D) (ontable C)
         (ontable A) (ontable B) (ontable D) (handempty)
         )
  (:goal (and (on D C) (on C B) (on B A)))
  (:metric minimize (total-cost))
)
```

V príklade 2 je vidieť v prvých riadkoch, že problém je označený menom `BLOCKS-WORLD-1` a je určený pre doménu `BLOCK-WORLD`. Nachádzajú sa tu štyri kocky A, B, C a D, ktoré ležia na stole. Cieľom je postaviť vežu, v ktorej kocka A je na spodku, nasleduje B a nakoniec kocka C. Metrika je v tom príklade

len symbolická, pretože každá akcia má rovnakú cenu. A samozrejme cieľom je minimalizovať celkovú cenu.

## B.3 História

Tak ako iné jazyky, tak aj PDDL prešiel od svojho vzniku vývojom. Všetky zmeny boli úzko späté so súťažou v plánovaní (IPC), ktorá sa od roku 1998 koná pravidelne každé dva roky.

### PDDL 1.2

Jazyk PDDL vznikol ako potreba jednotného štandardu pre súťaž v plánovaní IPC 1998[3]. Prvú verziu navrhol McDermott. Zodpovedala možnostiam vtedajších plánovačov, ktoré podporovali klasické plánovanie (STRIPS[15]).

### PDDL 2.0

Verzia 2.0 vznikla opäť pri príležitosti súťaže IPC 2000. Nepridalia žiadne nové funkcie, ale prispôsobila sa existujúcim plánovačom. Nezahrňuje doménové axiómy (Domain Axioms), obmedzenia (Constraints), hierarchické akcie (Hierarchical Actions) a prácu s číslami (Numerical Fluents).

### PDDL 2.1

V roku 2003 bol s verzou 2.1 štandard znova rozšírený o prácu s číslami (Numeric fluents), metriku plánu (Plan Metrics) a akcie so špecifikovanou dĺžkou trvania (Durative Actions).

### PDDL 2.2

Autor veľmi rozšíreného plánovača Fast-Forward[14] Jörg Hoffmann spolu s Stefanom Edelkampom a Michaelom Littmanom sa postarali o ďalšie rozšírenie PDDL: odvodene predikáty umožňujú manipuláciu s axiómami, časované počatočné literály, ktoré umožňujú ľahko reprezentovať deterministické exogénne udalosti.

### PDDL 3.0

Pri príležitosti IPC5[2] bola uvedená nová verzia jazyka PDDL s označením PDDL 3.0. Pribudli obmedzujúce podmienky (Constraints), ktoré musia byť splnené počas celého plánu. V praxi je v niektorých prípadoch výhodou, ak okrem hlavného cieľa je s rovnakými nárokmi splnený aj podcieľ. Pričom nie je nevyhnutné, aby bol podcieľ splnený vždy. V PDDL 3.0 preto pribudli preferencie (Preferences).

Tabuľka B.1: Prehľad verzií jazyka PDDL

Verzia	Rok	Autor	Funkcie
1.2	1998	Drew V. McDermott	STRIPS, Typing, Negativ Preconditions, Disjunctive Preconditions, Universal Preconditions, Quantified Preconditions, Conditional Effects
2.0	2000	Fahiem Bacchus	Odobrané: Domain Axioms, Constraints, Hierarchical Actions, Numerical Fluents
2.1	2002	Maria Fox, Derek Long	Numeric Fluents, Plan Metrics, Durativ Actions
2.2	2004	Jörg Hoffmann, Stefan Edelkamp, Michael Littmann	Derived Predicates, Timed Initial Literals
3.0	2006	Alfonso Gerevini, Derek Long	Goal Preferences, State Trajectory Constraints
3.1	2008	Malte Helmert	Objects fluents, Action cost



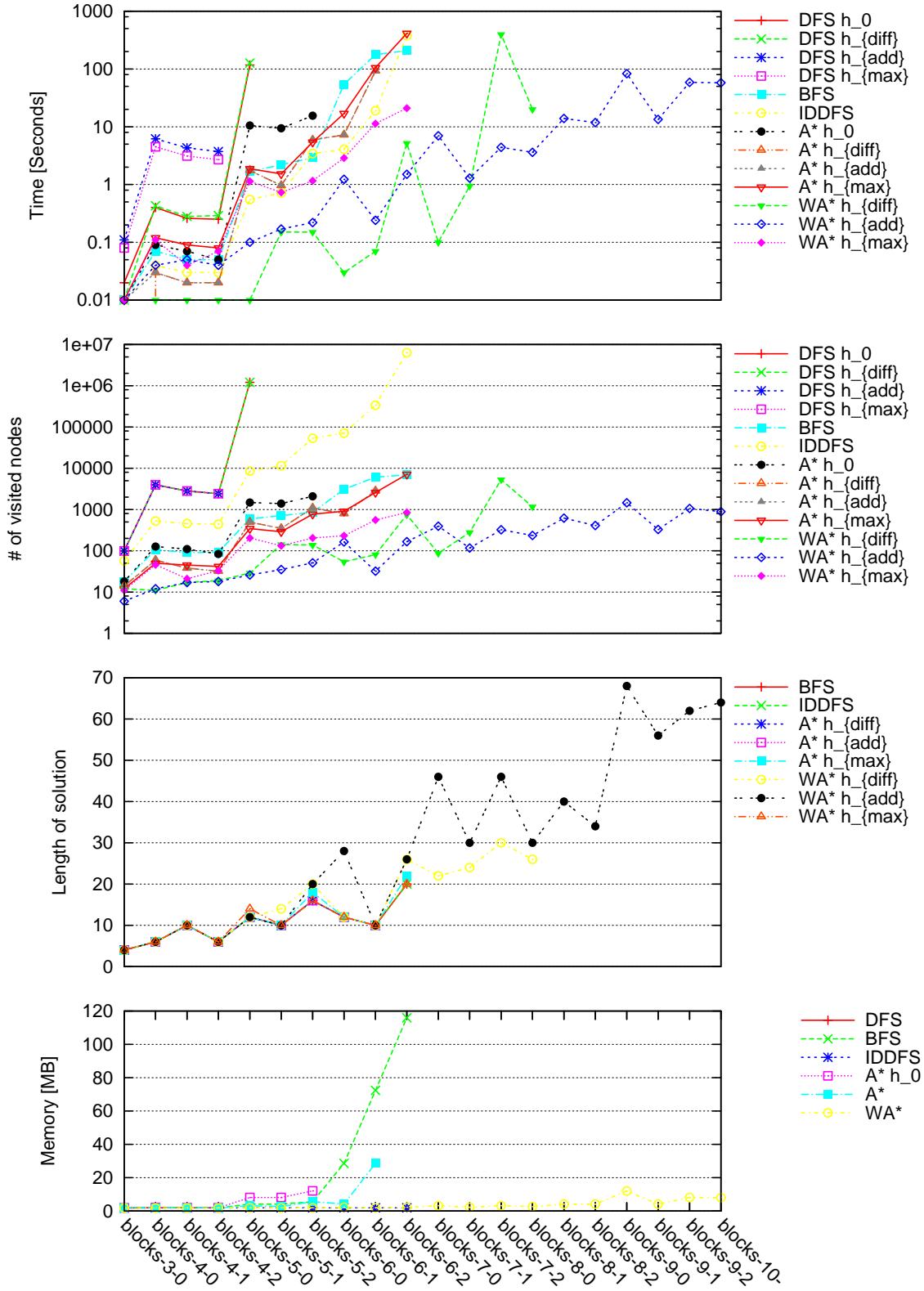
## Dodatok C

### Výsledky

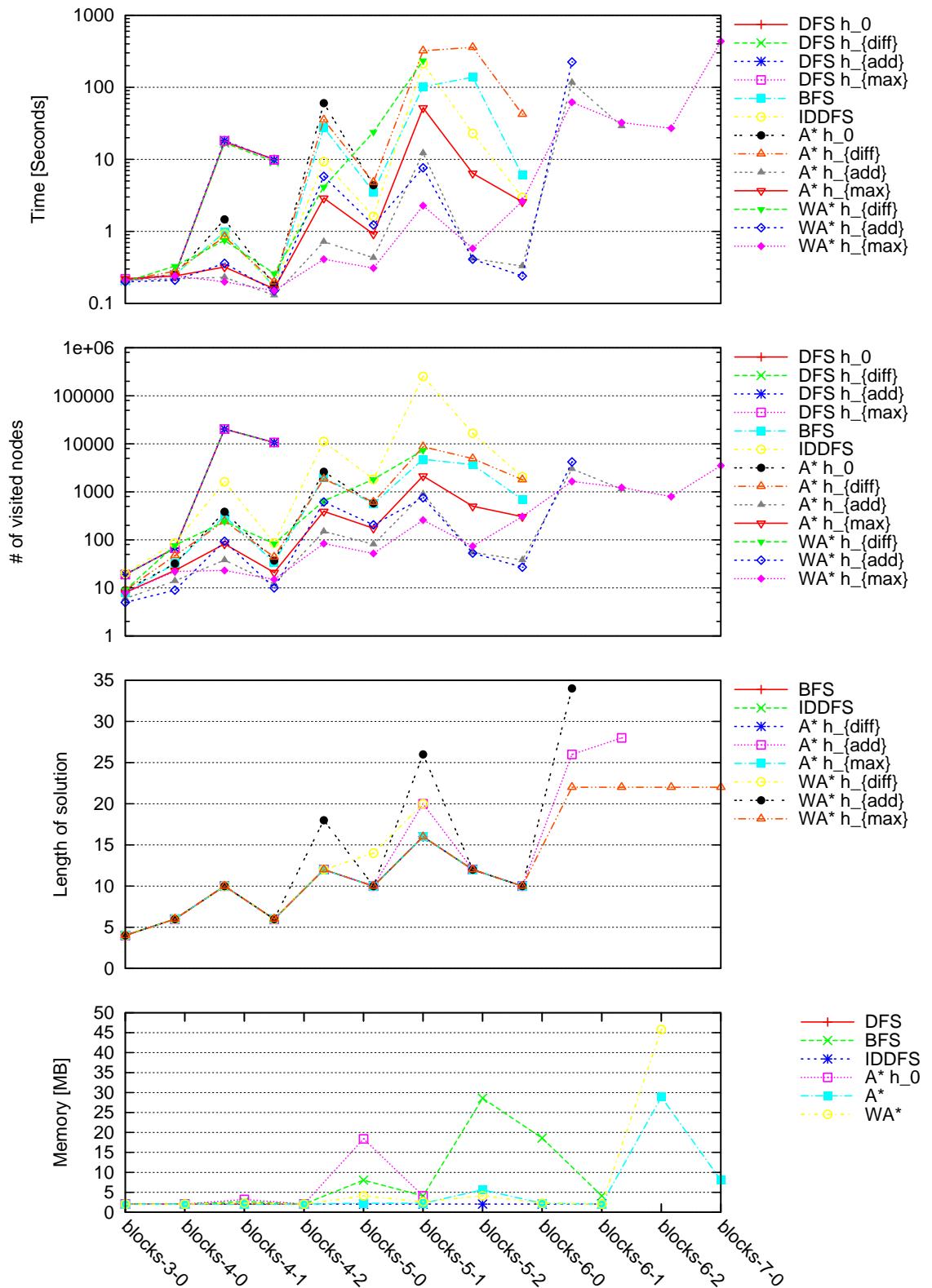
Na nasledujúcich stranách sú zobrazené grafy zo štatistiky behu plánovačov. Pre každú doménu a smer prehľadávania sú uvedené nasledujúce grafy: doba výpočtu, počet navštívených vrcholov, dĺžka plánu a množstvo spotrebovanej pamäte. Niektoré plánovače boli z grafov úmyselne vynechané, aby sa graf sprehľadnil. Poprípade boli plánovače z jednej triedy nahradené iba jedným reprezentatívnym. Ako príklad poslúži graf spotrebovanej pamäte, v ktorom sú všetky plánovače založené na prehľadávaní do hĺbky označené súborne DFS. Pričom rozdiel ich spotrebovanej pamäte bol minimálny.

Kompletnú štatistiku je možné nájsť na CD v adresári `output/`.

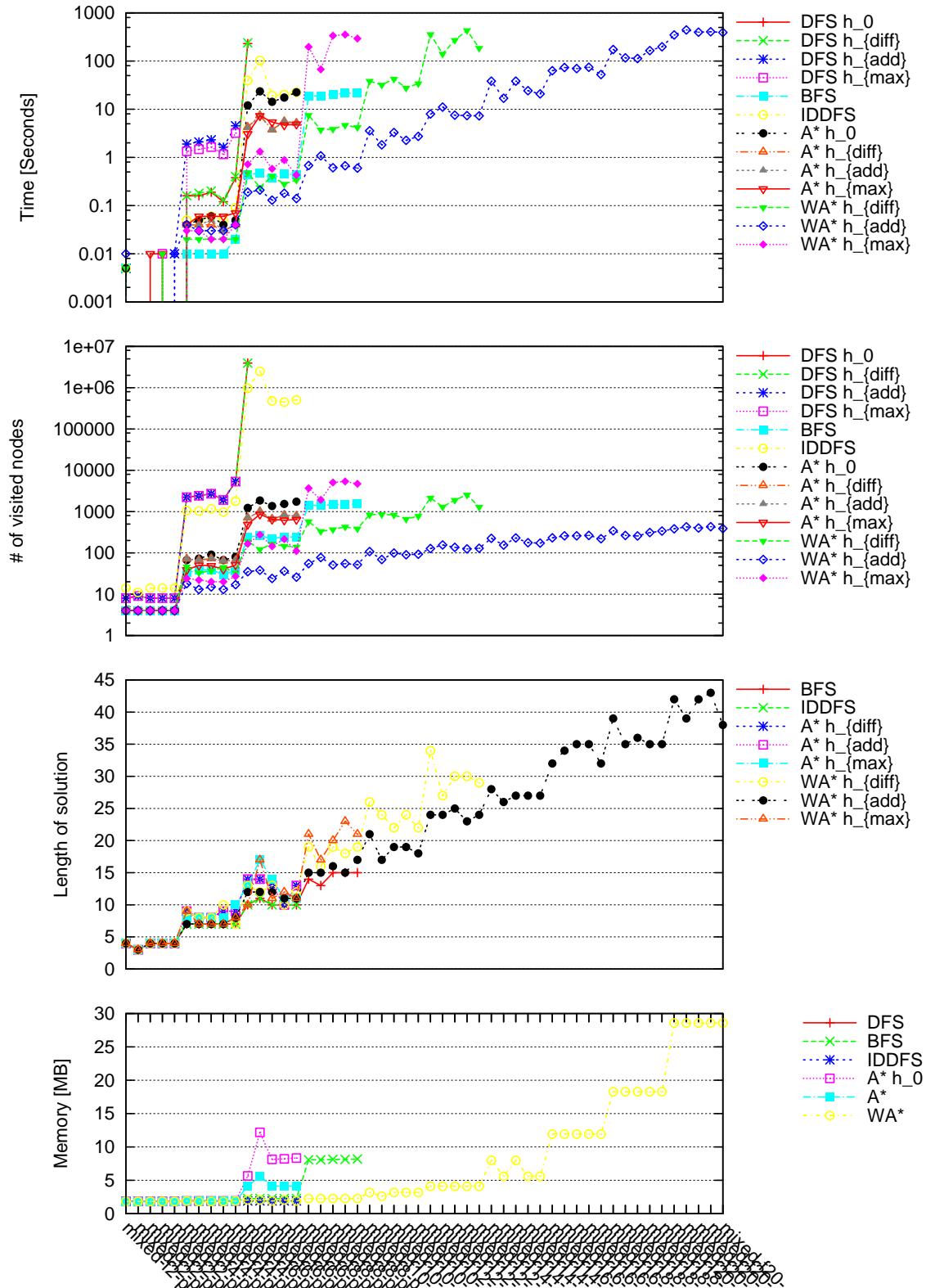
Graf C.1: Výsledky dopredného plánovania pre doménu Blocks World



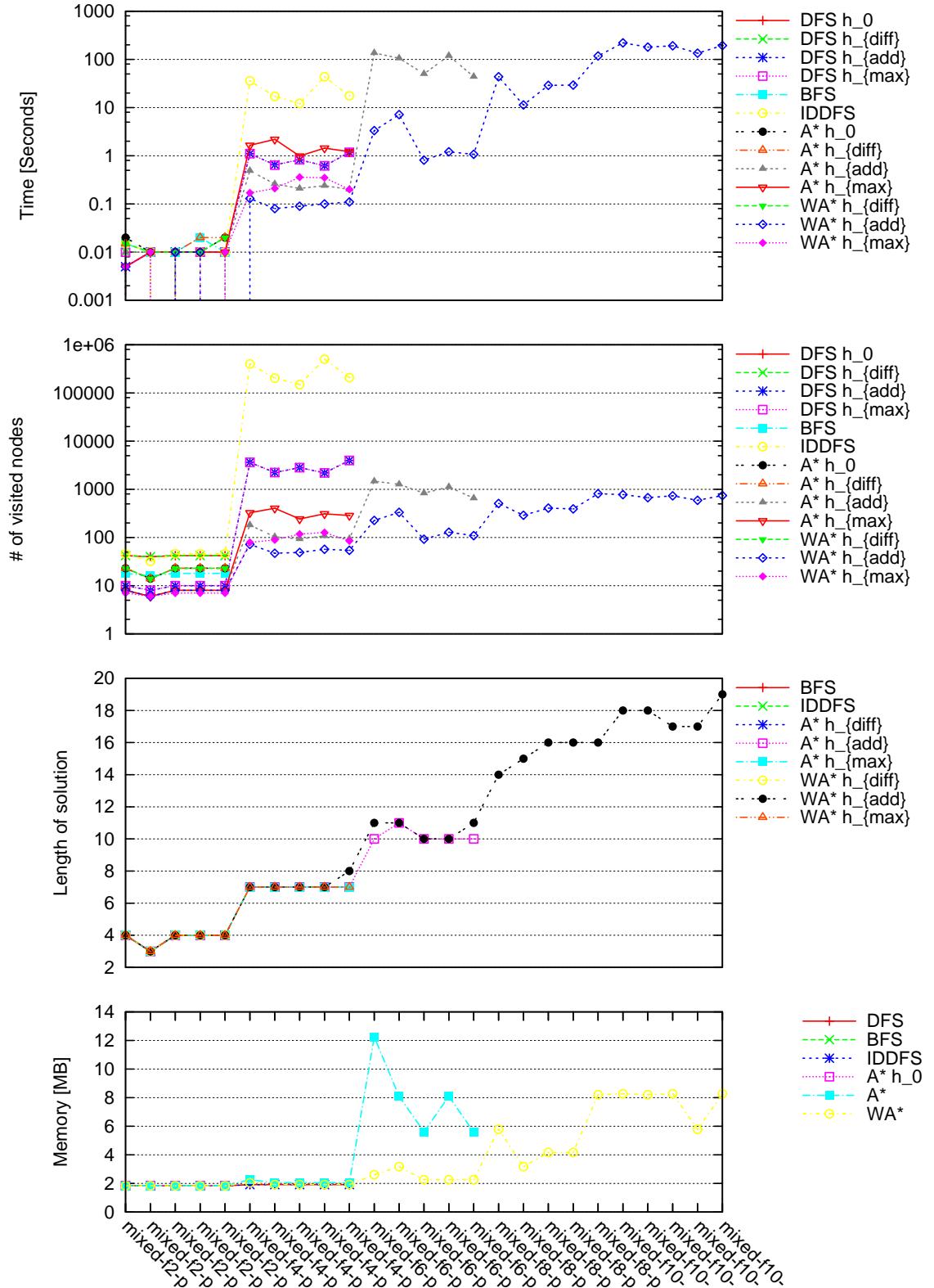
Graf C.2: Výsledky spätného plánovania pre doménu Blocks World



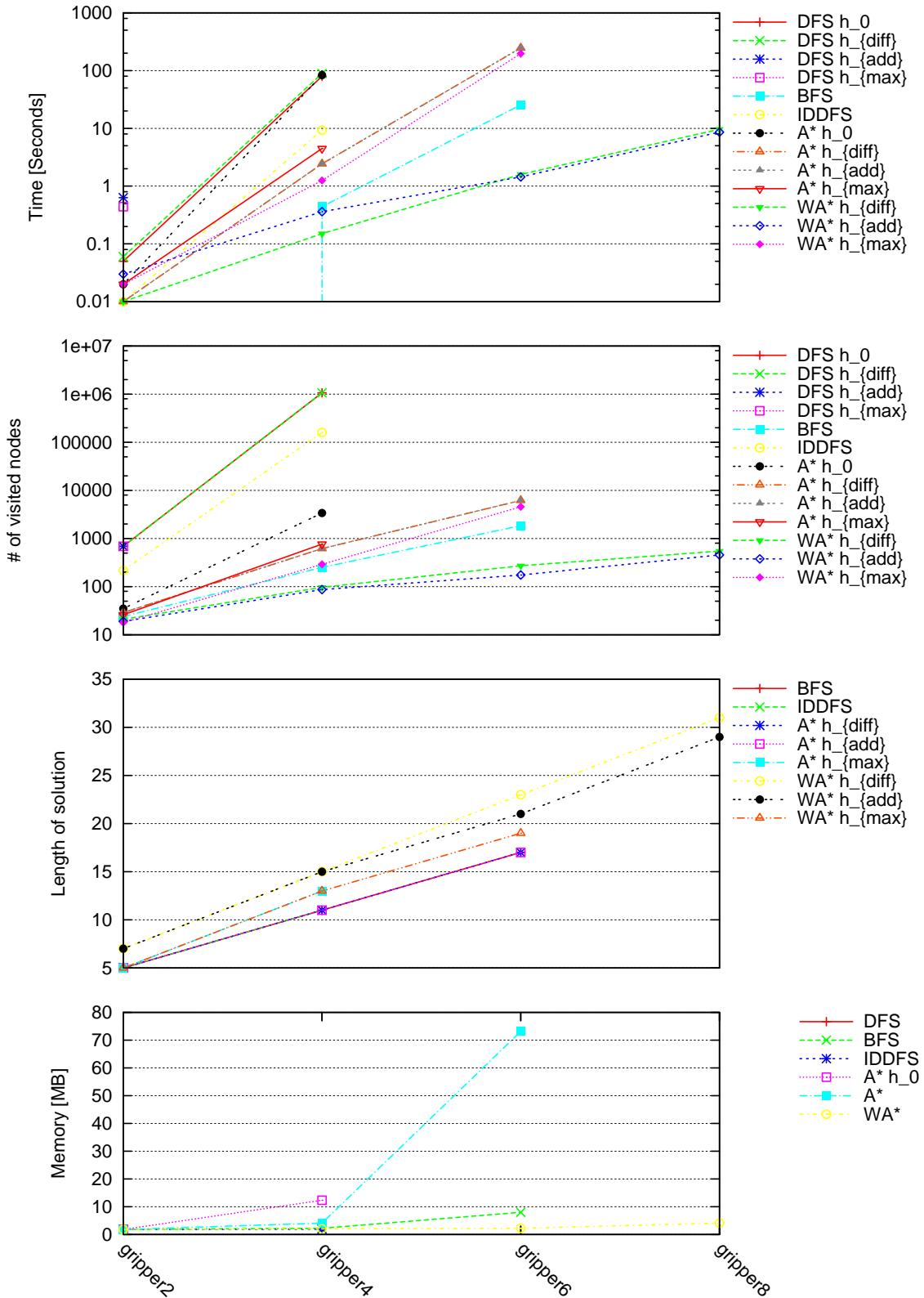
Graf C.3: Výsledky dopredného plánovania pre doménu Elevators



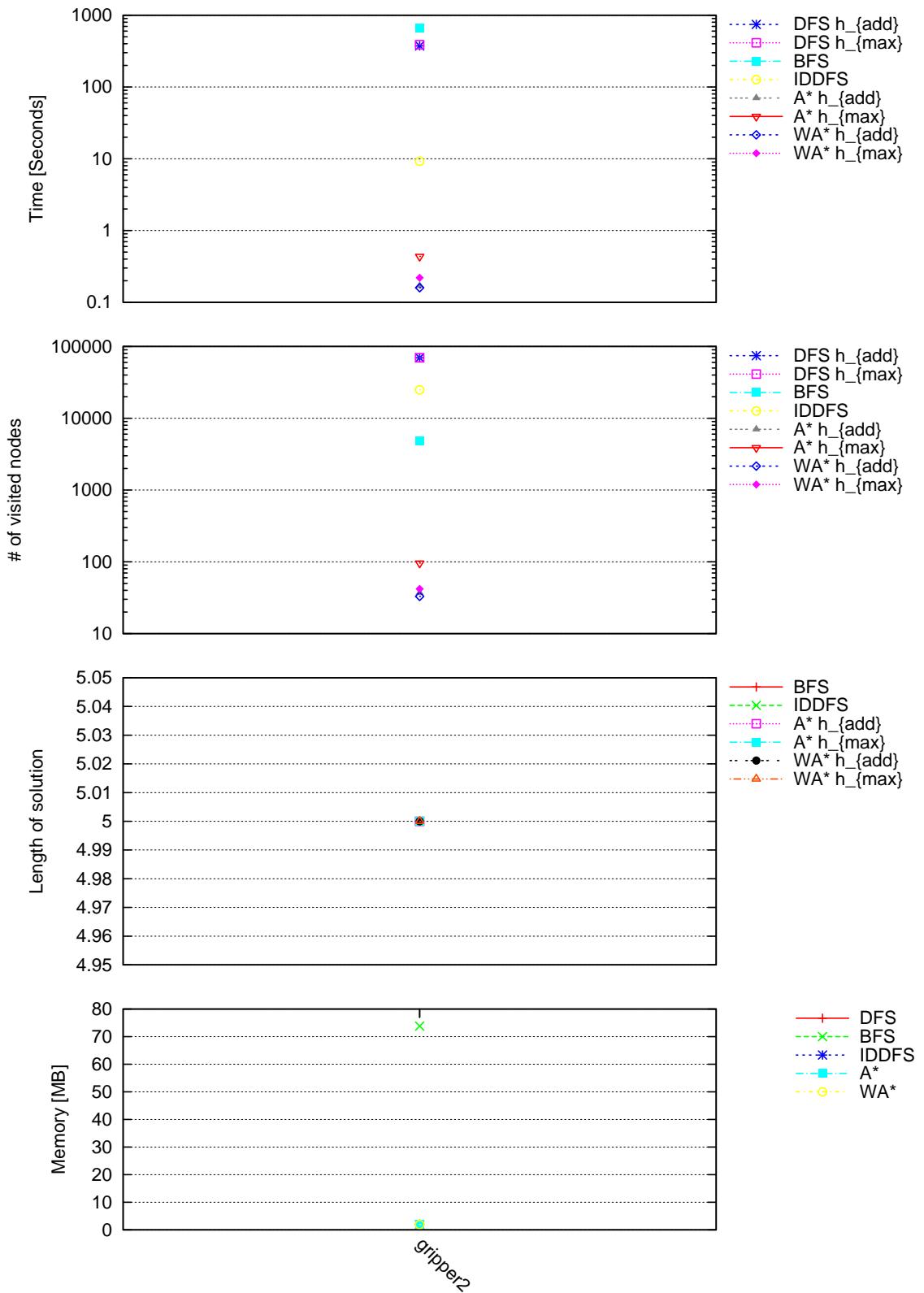
Graf C.4: Výsledky spätného plánovania pre doménu Elevators



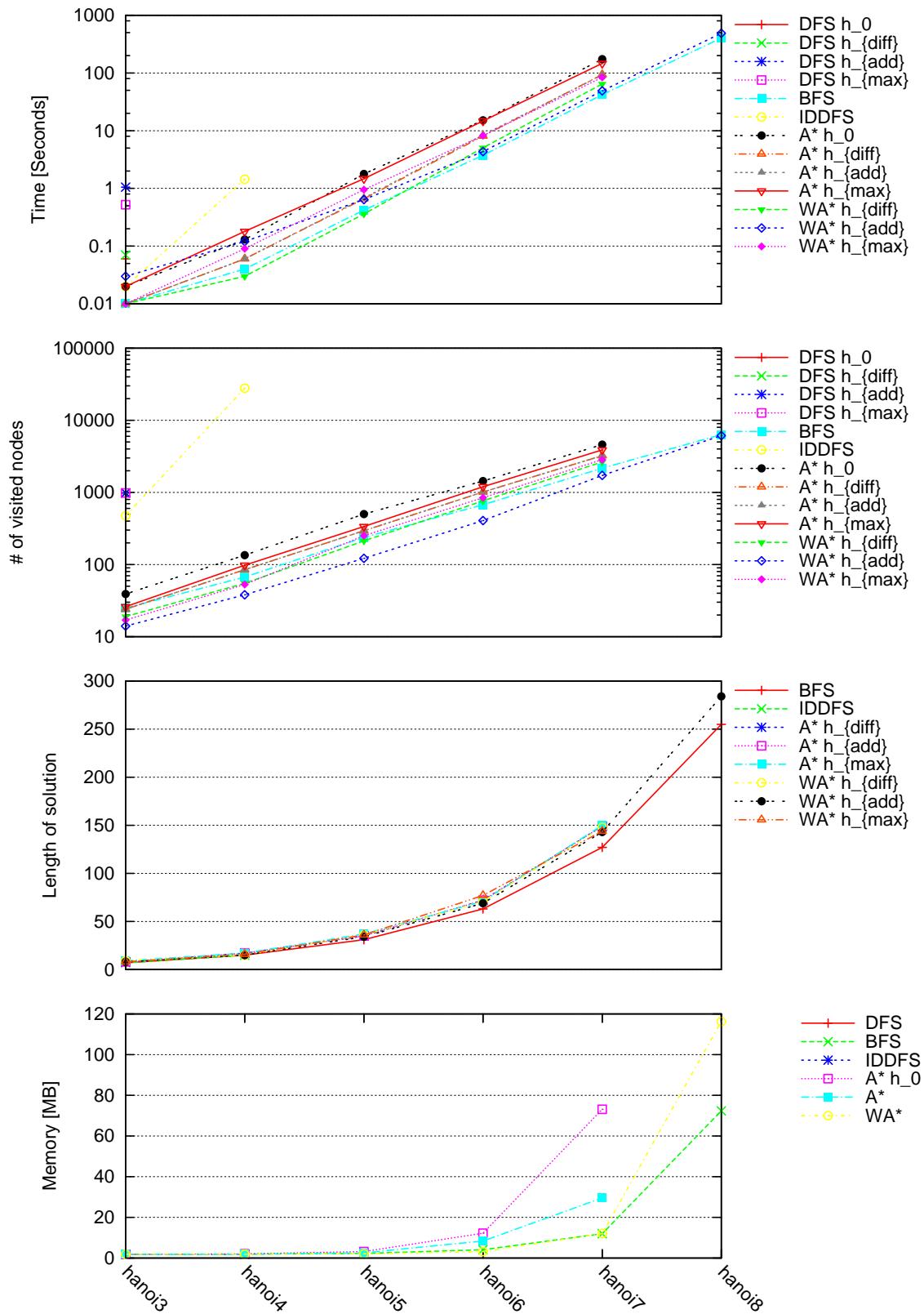
Graf C.5: Výsledky dopredného plánovania pre doménu Gripper



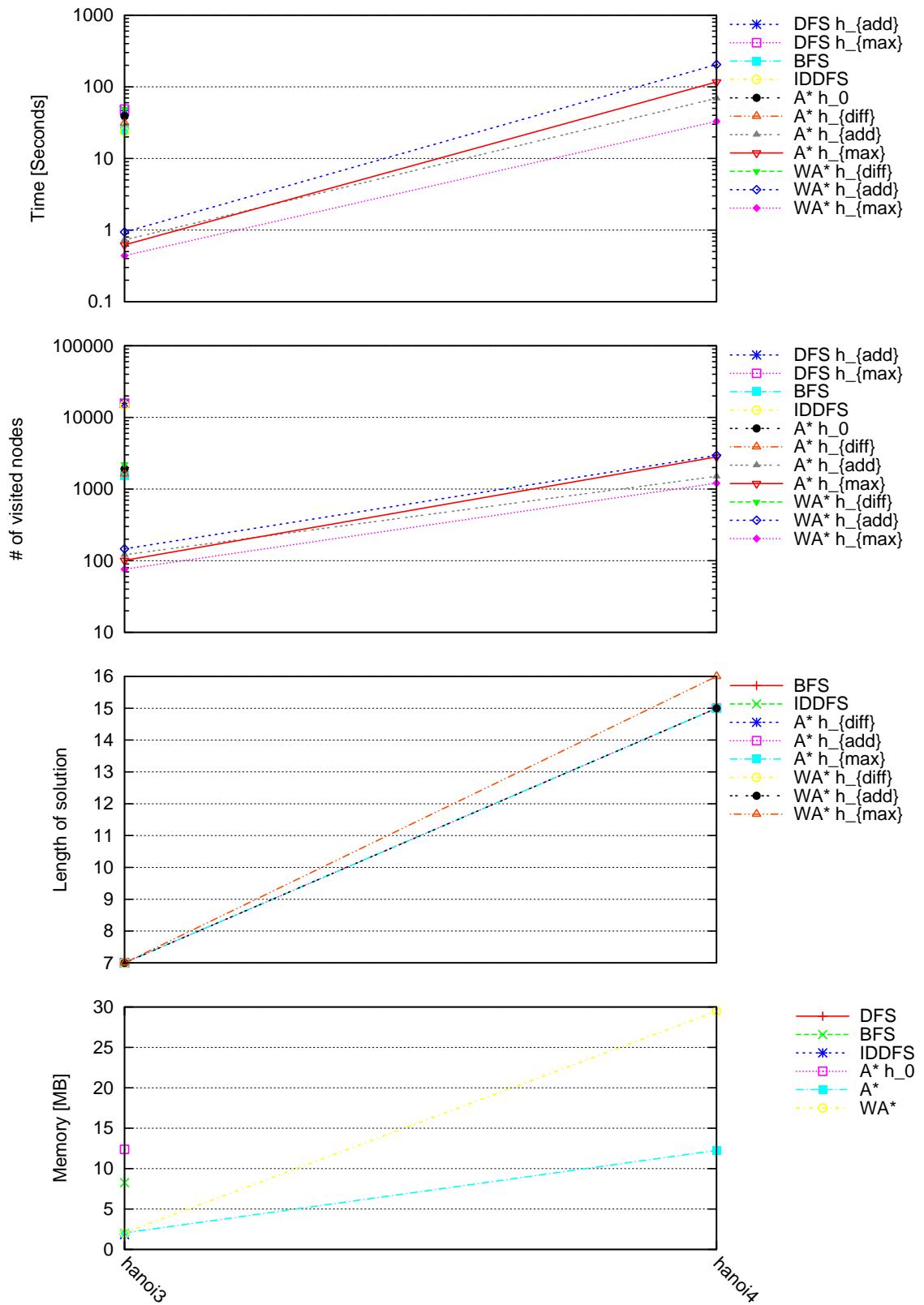
Graf C.6: Výsledky spätného plánovania pre doménu Gripper



Graf C.7: Výsledky dopredného plánovania pre doménu Hanoi



Graf C.8: Výsledky spätného plánovania pre doménu Hanoi





## **Dodatok D**

### **Obsah CD**

Na CD, ktoré je prílohou k tejto práci, je uložený adresár `planner` so všetkými zdrojovými kódmi. Súčasťou je aj skript `run.sh`, ktorý sekvenčne spustí všetky plánovače na všetkých plánovacích problémoch umiestnených v adresári `test/`. Taktiež je možné z výsledných dát vygenerovať grafy behu plánovačov pomocou skriptu `plot.sh`, ktoré je následne možno nájsť v adresári `charts`. Presný postup je uvedený v dokumentácii, ktorá sa nachádza v adresári `docs/`.