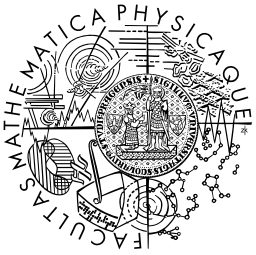


MATEMATICKO-FYZIKÁLNÍ FAKULTA
Univerzita Karlova

**BAKALÁŘSKÁ PRÁCE**

Jiří Kučera

# Učení plánovacích modelů

Katedra teoretické informatiky a matematické logiky

| | |
|---|---|
| Vedoucí bakalářské práce: | prof. RNDr. Roman Barták, Ph.D. |
| Studijní program: | Informatika |
| Studijní obor: | Programování |

Praha 2017

Prohlašuji, že jsem tuto bakalářskou práci vypracoval(a) samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V ..... dne ..... Podpis autora

Děkuji vedoucímu této práce, prof. RNDr. Romanovi Bartákovi, Ph.D., za veškerý čas a rady, které mi při psaní této práce věnoval.

Název práce: Učení plánovacích modelů

Autor: Jiří Kučera

Katedra: Katedra teoretické informatiky a matematické logiky

Vedoucí bakalářské práce: prof. RNDr. Roman Barták, Ph.D., Katedra teoretické informatiky a matematické logiky

Abstrakt: Programy generující plány vyžadují přesný popis plánovacího modelu. Vytvoření takového modelu je ale náročný proces, proto se vytváří postupy, které generují modely automaticky ze vzorových plánů. Mnoho z nich však klade speciální požadavky na učený model nebo nepracuje zcela přesně. V této práci představujeme nový přístup LOUGA (Learning operators using genetic algorithms), který se automaticky učí modely za pomoci genetických algoritmů. Na rozdíl od jiných přístupů neklade LOUGA žádné speciální požadavky na vlastnosti učených modelů a dokáže pracovat přesně i s malým množstvím predikátů ve vstupních plánech. Přístup v prvním kroku vygeneruje všechny možné dvojice operátor-predikát takové, že operátor může predikát přidat nebo odebrat ze světa. Každá tato dvojice je reprezentována jedním genem. V druhém kroku provádí evoluci jedinců, dokud nenalezne takový model, který vysvětluje vstupní data bez chyb. V experimentech jsme ověřili, že LOUGA pracuje znatelně přesněji a rychleji než existující přístup ARMS.

Klíčová slova: plánování modelování učení

Title: Learning planning models

Author: Jiří Kučera

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: prof. RNDr. Roman Barták, Ph.D., Department of Theoretical Computer Science and Mathematical Logic

Abstract: In AI planning, planners typically require a precise description of the input model. Creation of such model is a difficult task, so methods that automatically generate models from input plans are also created. However, a lot of them make assumptions about the model or are imprecise. In this thesis, we present a new method called LOUGA (Learning operators using genetic algorithms), which uses genetic algorithms to learn models. Unlike other methods, LOUGA does not make any assumptions about the model and works precisely even with small amount of predicates in input plans. In the first step, LOUGA generates all such pairs operator-predicate, that the operator can add or remove the predicate from the world. Every such pair is represented by one gene. In the second step, evolution is being performed until such model that explains input data without errors is found. In this paper, we have also proved empirically that LOUGA works faster and more precisely than existing method ARMS.

Keywords: planning modeling learning

# Obsah

**Úvod** **3**

**1 Základní pojmy a definice** **4**
1.1 Plánovací model . . . . . 4
1.2 Plánovací problém . . . . . 5
1.3 Genetický algoritmus . . . . . 5

**2 Popis řešeného problému** **7**
2.1 Neformální popis problému . . . . . 7
2.2 Problém řešený přístupem LOUGA . . . . . 8
2.3 Rozdíly v zadáních pro ostatní automatické přístupy . . . . . 8

**3 Existující přístupy** **9**
3.1 ARMS . . . . . 9
3.2 AMAN . . . . . 10
3.3 Opmaker . . . . . 11
3.4 Opmaker2 . . . . . 11
3.5 LOCM . . . . . 12
3.6 LOCM2 . . . . . 14
3.7 Shrnutí . . . . . 16

**4 Přístup LOUGA** **17**
4.1 Vstupní data . . . . . 17
4.2 Příprava evoluce . . . . . 17
4.3 Fitness funkce . . . . . 21
4.4 Operátory genetického algoritmu . . . . . 22
4.5 Dodatečné generování precondition seznamů . . . . . 23
4.6 Generování modelu predikát po predikátu . . . . . 23

**5 Experimentální výsledky** **26**
5.1 Metodika měření . . . . . 26
5.2 Přehled domén . . . . . 27
5.3 Implementace ARMS . . . . . 27
5.4 Porovnání s ARMS . . . . . 28
5.5 Evoluce vs. evoluce predikát po predikátu . . . . . 31
5.6 Svět bez typů . . . . . 31

**Závěr** **33**

**Seznam použité literatury** **34**

**Přílohy** **35**

**A Uživatelská dokumentace** **36**
A.1 Přehled UI . . . 36
A.2 Práce s jednotlivými přístupy . . . 36
A.2.1 LOUGA . . . 36
A.2.2 ARMS . . . 39
A.2.3 LOCM . . . 40
A.2.4 LOCM2 . . . 41

**B Syntaxe používané verze jazyka PDDL** **42**
B.1 Definice modelu . . . 42
B.2 Definice světů a plánů . . . 43

**C Programátorská dokumentace** **46**
C.1 Projekt ModelLearner . . . 46
C.1.1 Implementace LOUGA . . . 46
C.1.2 Implementace ARMS . . . 47
C.1.3 Implementace LOCM . . . 48
C.1.4 Implementace LOCM2 . . . 48
C.2 Projekt PlanGenerator . . . 48
C.3 Knihovna GeneralPlanningLibrary . . . 48

**D Dokumentace ke generátoru plánů** **50**
D.1 Popis generátoru . . . 50
D.2 UI generátoru . . . 50

**E Obsah CD** **52**

# Úvod

Plánování představuje jeden z oborů umělé inteligence. Zabývá se otázkou, jak nalézt takovou posloupnost akcí (tuto posloupnost nazýváme plán), po jejímž provedení se svět dostane do stavu splňujícího předem dané podmínky. Na hledání plánů existuje již velké množství přístupů a stále se vyvíjejí lepší, každý ale od uživatele vyžaduje přesný model popisující, jak které akce mění stav světa. V praxi je často složité vytvořit přesný popis velkých plánovacích modelů, vyvíjejí se proto i přístupy, které se snaží toto dělat za uživatele automaticky.

Cílem této práce je navrhnout a implementovat novou metodu pro strojové učení se plánovacích modelů. Součástí práce je i prostředí, ve kterém lze vytvořený přístup používat, sledovat jeho průběh a porovnávat jeho výsledky s výsledky jiných přístupů. Část této práce tedy bude spočívat i v implementaci několika existujících metod, které taktéž řeší problém učení se plánovacích modelů ze vzorových plánů a následného experimentálního porovnání výkonnosti nového přístupu s některým z již používaných, řešícím podobné zadání.

Výsledkem této práce je LOUGA (Learning operators using genetic algoritms), nový přístup, který pro učení používá genetické algoritmy.

V rámci práce vznikly i dvě aplikace. První z nich zvládá zobrazovat a upravovat vstupní data v PDDL formátu a testovat, zda v nich nejsou chyby. Obsahuje vlastní implementace čtyř přístupů: ARMS, LOCM, LOCM2 a LOUGA, vypisuje uživateli zevrubný průběh jejich práce a zobrazuje výsledky jejich běhů. Druhá aplikace slouží pro generování základních vstupů pro testování přístupů.

V první části textu definujeme pojmy a struktury, se kterými budeme v práci pracovat a přesně si definujeme problém, který bude výsledný přístup řešit. V další části si zevrubně představíme již existující přístupy, které slouží k řešení podobných problémů. Následně podrobně rozebereme nový přístup LOUGA a nakonec experimentálně ověříme, že se dokáže naučit i netriviální modely, a porovnáme jeho výkon s již existujícím přístupem.

# 1. Základní pojmy a definice

V této kapitole definujeme základní pojmy, se kterými budeme v práci pracovat.

## 1.1 Plánovací model

Plánovací model je zjednodušený popis fungování skutečného světa. Popis modelu se skládá z definic predikátů a operátorů. Pomocí predikátů uživatel popisuje, jak vypadá stav světa. Akce slouží k přecházení mezi jednotlivými stavy, každá akce tedy ze stavu nějaké predikáty maže a nějaké predikáty do něj přidává. Operátory jsou obecné popis akcí.

Formálně můžeme plánovací model definovat jako trojici (T, P, O), kde

- T je množina typů objektů,
- P je množina definicí typů predikátů — definice predikátu se skládá z jména predikátu a seznamu parametrů s možnými typy,
- O je množina operátorů. Každý operátor se skládá z:
  - unikátního jména operátoru
  - seznamu parametrů s možnými typy objektů
  - precondition seznamu - seznamu predikátů, které ve světě musí platit, než je akce provedena
  - add seznamu - seznamu predikátů, které akce do světa přidá
  - del seznamu - seznamu predikátů, které akce ze světa smaže

  Instanci operátoru s konkrétními argumenty nazýváme akcí.

Svět je instance modelu se seznamem používaných objektů. Stav ve světě je množina predikátů. Plán je posloupnost akcí. Součástí popisu plánu můžou být i seznamy predikátů v některých stavech.

Add, del a precondition seznamy pro akci $a$ značíme $add_a$, $del_a$ a $pre_a$. Nechť $S = \{p_1, p_2, ...p_n\}$ je stav skládající se z predikátů $p_1$ až $p_n$. Akce $a$ je aplikovatelná na stav $S$ právě tehdy, pokud $pre_a \subset S$. Aplikováním akce $a$ na $S$ získáme nový stav $T$ definovaný jako $T = (S - del_a) \cup add_a$.

Mějme například model domény *Blocksworld*. Tato standardní doména popisuje svět, ve kterém jeřáb staví věže z kostek. Používá 4 operátory: jeden pro položení kostky na zem, další pro zvednutí kostky ze země, třetí pro postavení kostky na jinou kostku a poslední pro zvednutí kostky z jiné kostky. První 2 operátory můžou být implementovány například takto (specifikace používaného jazyka viz. Příloha I - Uživatelská dokumentace):

```
(:action pickup
  :parameters (?ob)
  :precondition (and (clear ?ob) (on-table ?ob) (arm-empty))
  :effect (and (holding ?ob) (not (clear ?ob)) (not (on-table ?ob))
```

```
              (not (arm-empty))))

(:action putdown
  :parameters  (?ob)
  :precondition (holding ?ob)
  :effect (and (clear ?ob) (arm-empty) (on-table ?ob)
               (not (holding ?ob))))
```

a stav s predikáty:

```
(clear a)
(on-table a)
(holding b)
```

Akce (*pickup a*) ani (*pickup b*) nemůžeme provést, protože ve stavu není predikát (*arm-empty*). Akci (*putdown a*) také nemůžeme provést, protože chybí predikát (*holding a*). Můžeme provést pouze akci (*putdown b*). Po aplikování této akce se svět dostane do stavu:

```
(clear a)
(on-table a)
(clear b)
(on-table b)
(arm-empty)
```

Z tohoto stavu nelze provést akce (*putdown b*) ani (*putdown a*), ale můžeme provést akce (*pickup a*) či (*pickup b*).

## 1.2 Plánovací problém

Plánovací problém je definován jako ($s_0$, $g$, $M$), kde $s_0$ je počáteční stav, $g$ je množina predikátů, které mají být přítomny v koncovém stavu a $M$ je plánovací model. Cílem plánovacího programu je nalézt platnou posloupnost akcí, která začíná stavem $s_0$ a končí stavem, který obsahuje všechny predikáty z $g$.

## 1.3 Genetický algoritmus

Přístup LOUGA, který je výsledkem této práce, ve svém jádru používá genetický algoritmus [1]. Proto si nyní popíšeme, jak takový algoritmus běžně funguje.

Genetický algoritmus je typ optimalizačního algoritmu. Pracuje s množinou jedinců, kteří reprezentují částečná řešení daného problému. V průběhu svého běhu algoritmus tyto jedince mírně upravuje, kříží je vzájemně mezi sebou a snaží se najít co nejlepší řešení problému.

Prostředí genetického algoritmu můžeme definovat jako (P, F, O), kde

- P je populace, tedy množina aktuálních jedinců. Jedinec je typicky pole čísel nebo boolovských hodnot. Toto pole nazýváme taky genomem, jednotlivé položky v poli jsou geny.

- F je fitness funkce, tedy funkce, která každému jedinci přiřazuje hodnotu určující jeho kvalitu,
- O je množina operátorů, které nějak upravují genomy jednoho či více jedinců. Typicky se používají mutace a křížení. Mutace jsou operátory, které s určitou pravděpodobností mění hodnoty jednotlivých genů. Křížení jsou pak operátory, které vezmou 2 či více jedinců a vytvoří nové jedince kombinací jejich genomů.

Běh genetického algoritmu můžeme shrnout následovně:

**Algoritmus 1** Genetický algoritmus

```
1: náhodně vytvoř počáteční populaci P(0)
2: t = 0
3: while není splněna ukončující podmínka do
4:     vyber z P(t) jedince ke křížení
5:     zkřiž mezi sebou vybrané jedince
6:     náhodně mutuj některé jedince P(t)
7:     vytvoř P(t+1) z P(t) a nově vytvořených jedinců.
8:     t = t + 1
9: end while
```

Algoritmus bývá typicky ukončen, když nalezne jedince s dostatečnou fitness nebo po určitém počtu generací.

# 2. Popis řešeného problému

V této kapitole přesně definujeme problém, který řeší přístupy zmíněné v této práci. Jelikož se ale jednotlivá zadání mírně liší, tak nejprve definujeme problém, který řeší LOUGA, a pak rozebereme, jak se liší zadání ostatních přístupů.

## 2.1 Neformální popis problému

Neformálně můžeme říci, že cílem učícího se programu je zjistit, jakými způsoby je možné v daném modelu měnit stav světa. Program dostane na vstupu vzorové posloupnosti provedených akcí a snaží se odhadnout, co se světem udělaly konkrétní akce. Součástí vstupu bývají často i informace o tom, že v daném stavu byl vypozorován nějaký konkrétní predikát. Tyto informace bývají typicky potřeba, aby měl učící se program nějaké vodítko při hledání správných definic operátorů. Některé přístupy si ale vytvářejí vlastní reprezentaci světa, a tak s původními predikáty vůbec nepracují a tyto informace nepotřebují.

Například můžeme mít zadání pro model domény *Blocksworld*:

```
(:state
        (clear a)
        (on-table a)
        (arm-empty)
        (clear b)
        (on-table b))
(pickup a)
(stack a b)
(:state
        (on a b)),
```

což je plán s dvěma akcemi (*pickup a*) a (*stack a b*), v počátečním stavu bylo vypozorováno 5 predikátů a v koncovém stavu pouze predikát (*on a b*). V ideálním případě by měl učící se program najít originální tvary operátorů, nicméně z hlediska programu jsou tyto řešení nerozlišitelné od ostatních řešení, které korektně popisují vzorové plány, a tak cílem přístupů bývá najít libovolné korektní tvary operátorů. V tomto případě máme velmi malá vstupní data, obstály by tedy i operátory, které by vypadaly například takto:

```
(:action pickup
        :parameters (?ob)
        :precondition (arm-empty)
        :effect (not (arm-empty)))
(:action stack
        :parameters (?ob ?ob2)
        :precondition (and (clear a) (clear b))
        :effect (and (holding b) (not clear a) (on a b)))
```

Tyto operátory z pohledu popisu domény *Blocksworld* nedávají smysl, nicméně z hlediska učícího se programu jsou korektním řešením. Že se toto řešení odlišuje od původního tvaru operátorů není problém učícího se programu, ale víceznačnosti vstupních dat.

## 2.2 Problém řešený přístupem LOUGA

Formálně definujeme problém učení se plánovacích modelů jako dvojici ($P$, $M$), kde $P$ je vzorová množina plánů a $M$ je částečný plánovací model, ve kterém chybí precondition, add a del seznamy operátorů. V každém plánu $P$ musí být všechny provedené akce a počáteční stav, z ostatních stavů může být vypozorována jen část predikátů, nebo můžou úplně chybět (nesmí být ale být uvedeny žádné predikáty, které ve stavu nebyly). Cílem učícího programu je nalézt pro každý operátor takové seznamy, které korektně popisují vstupní plány. Tedy pokud vezmeme libovolný vzorový plán, začneme v jeho počátečním stavu a další stavy získáváme tak, že aplikujeme akce podle výsledných definic operátorů, tak pro každou akci $a$ musí platit, že $pre_a \subseteq S$, $add_a \cap S = \emptyset$ a $del_a \subseteq S$, kde $S$ je stav těsně před provedením akce $a$. Navíc každý stav musí obsahovat všechny predikáty odpozorované v odpovídajícím stavu vstupního plánu a pokud je daný stav označený jako úplný, nesmí v něm být přítomny žádné jiné predikáty.

Podmínky $add_a \cap S = \emptyset$ a $del_a \subseteq S$ jsou silnější podmínky než je nezbytně nutné pro obecný učící se přístup. Striktně vzato, model může do světa přidat predikát, který už v něm je, nicméně v dobře vytvořených modelech by to nemělo nastávat. Pro přístup LOUGA jsou tyto předpoklady nezbytné, neboť jich hojně využívá při hodnocení aktuálně vygenerovaných modelů.

## 2.3 Rozdíly v zadáních pro ostatní automatické přístupy

V této části rozebereme, jak se liší problém řešený ostatními přístupy zmíněnými v této práci. Tyto přístupy podrobně popisujeme v kapitole 3

ARMS [2] se liší pouze v tom, že v každém plánu vyžaduje cílové predikáty, jinak používá stejné zadání jako LOUGA.

AMAN [3] přebírá zadání od ARMS a navíc povoluje, aby některé z akcí ve vstupních plánech byly špatně rozpoznané, tedy že ve skutečnosti mohly být provedeny jiné akce, než jaké jsou vypsány na vstupu. Jeho cílem je kromě vytvoření seznamů i nalezení a opravení těchto chybných akcí.

LOCM [4] a LOCM2 [5] tvoří vlastní reprezentaci světa, takže nevyžadují v plánech žádné rozpoznané stavy a ani v modelu nepotřebují seznam definic typů predikátů. Jejich cílem je kromě vytvoření seznamů pro operátory i vytvoření nových predikátů, které budou reprezentovat stavy jednotlivých objektů. Pracují dobře pouze pro modely, jejichž objekty jdou popsat konečnými automaty, nicméně teoreticky dokáží pracovat s libovolným modelem.

Opmaker2 [6] problém definuje jako trojici ($P$, $M$, $I$), kde $P$ je množina plánů s počátečními a koncovými stavy, $M$ je částečný model bez add, del a precondition seznamů operátorů, ve kterém má každý parametr každého operátoru jasně řečeno, jestli po provedení dané akce mění stav nebo ne, a $I$ je množina logických výroků, které musí být platné v každém stavu světa. Navíc vyžaduje, aby všechny objekty šly popsat konečným automatem. Cílem je nalézt takové seznamy, které vyhovují vstupním plánům a výrokům.

# 3. Existující přístupy

V této kapitole shrneme fungování několika již existujících přístupů, které se učí plánovací modely ze vzorové množiny plánů. Zaměříme se celkem na 6 přístupů.

ARMS [2] řeší úlohu převodem na vážený MAX-SAT problém, AMAN [3] se snaží pracovat se šumem ve vstupních plánech a převádí úlohu na problém hledání co nejlepších hodnot dvou vektorů čísel. Opmaker [7] je poloautomatický nástroj pomáhající uživatelovi při vytváření doménového modelu. Jeho vylepšení Opmaker2 [6] generuje domény za pomoci logických výroků popisujících vlastnosti hledaného modelu, které vyžaduje na vstupu spolu s plány. LOCM [4] sleduje, v jakém pořadí můžou být s jednotlivými objekty prováděny akce a ze získaných dat tvoří pro každý typ objektu konečný automat. LOCM2 [5] tuto metodu zlepšuje tím, že netvoří jeden, ale skupinu automatů. Každý z nich popisuje objekt z jiného úhlu pohledu.

Všechny tyto přístupy podrobněji rozebereme a na závěr shrneme jejich základní vlastnosti.

## 3.1 ARMS

ARMS [2] je pravděpodobně nejznámějším učícím se přístupem. Učí se operátory převedením problému na vážený MAX-SAT problém.

ARMS dostane na vstupu množinu plánů s počátečními stavy, cílovými predikáty a případnými částečně rozpoznanými mezistavy a vytvoří logickou formuli skládající se z klauzulí 4 typů.

Klauzule prvního typu popisují formální požadavky na pre, add a del seznamy každého operátoru. Akce by neměla přidávat predikát, který vyžaduje ve výchozím stavu a měla by vyžadovat predikáty, které se chystá smazat.

- $pre_o \cap add_o = \emptyset$
- $del_o \subset pre_o$

Klauzule druhého typu vysvětlují výskyty predikátů v mezistavech: pokud predikát nebyl v počátečním stavu plánu, musela ho nějaká předchozí akce do světa přidat a současně ho poslední akce před stavem nemohla odstranit.

- $p \in add_1 \vee p \in add_2 \vee \ldots \vee p \in add_i$
- $p \notin del_i$

Pro vytváření klauzulí třetího typu ARMS prochází plány a hledá dvojice predikát-akce ($p$,$a$) takové, že $p$ je pozorováno ve světě před provedením akce $a$. Předpokládá, že pokud se tato akce objevuje často, pravděpodobně bude $p$ potřeba k provedení $a$.

- $p \in pre_a$

Při vytváření klauzulí posledního typu předpokládá, že žádná akce ve vstupních plánech není zbytečná. Každá akce do světa přidává buď cílový predikát nebo předpoklad nějaké jiné akce. Současně každý predikát v pre seznamu každé akce musí být buď v počátečním stavu plánu nebo být přidán nějakou předcházející akcí a nesmí být smazán žádnou akcí mezi nimi. Pro reálné použití by ale byla takto definovaná pravidla příliš dlouhá, a proto se ARMS zaměřuje pouze na páry akcí (a,b), které se často vyskytují po sobě. Připouští i možnost, že druhá akce může sloužit k tomu, že přidává zpátky do světa predikát, která předchozí akce smazala.

- $\exists p : p \in (pre_a \cap pre_b) \wedge p \notin del_a$
- $\exists p : p \in (add_a \cap pre_b)$
- $\exists p : p \in (del_a \cap add_b)$

V dalším kroku algoritmus jednotlivým klauzulím udělí váhy. Klauzule 1. a 2. typu jsou považovány za důležitější, proto mají vysoké váhy, konkrétní hodnoty jsou empiricky určovány pro každou doménu zvlášť. Klauzule zbývajících typů mají váhy dané frekvencemi výskytu odpovídajících párů.

Hlavní výhodou tohoto přístupu je jeho univerzálnost. Neklade na učené modely žádné zvláštní požadavky jako LOCM/LOCM2, ani o nich nevyžaduje žádné dodatečné informace jako Opmaker a Opmaker2. Jeho slabou stránkou je naopak to, že nalezené modely jsou často mírně chybné. Tedy že některé akce ve vstupních plánech nemají splněné všechny předpoklady nebo že v některých stavech chybí predikáty, které byly ve vstupních datech.

## 3.2 AMAN

Přístup AMAN [3] pracuje s plány, ve kterých můžou být některé akce špatně rozpoznané. Snaží se odhadnout, které akce jsou chybné a najít správný plánovací model.

Jeho vstupem jsou potenciálně chybně rozpoznané posloupnosti akcí s počátečními a koncovými stavy.

AMAN převádí učící problém na problém nalezení co nejlepších hodnot dvou vektorů čísel. První vektor určuje váhy modelů, se kterými přístup pracuje. Hodnoty v druhém vektoru určují váhy 10 vlastností, které slouží k počítání pravděpodobnosti $p(a|a_0, s, m)$, tedy pravděpodobnosti, že byla provedena akce $a$ za předpokladu, že ve vstupním plánu byla pozorována akce $a_0$, předchozí stav byl stav $s$ a skutečný model je $m$. Těmito vlastnostmi jsou například aplikovatelnost akcí $a$ a $a_0$ na $s$ či podobnost jejich add, del a precondition seznamů.

V prvním kroku AMAN nejprve vygeneruje množinu všech možných potencionálních modelů. To provádí tak, že vezme všechny dvojice ($o$, $p$) takové, že operátor $o$ může použít predikát $p$ ve svých add, del a precondition seznamech, a pomocí těchto dvojic generuje všechny možnosti, jak můžou seznamy jednotlivých operátorů vypadat. Ze všech takovýchto modelů vybere pouze ty, které splňují následující podmínky:

- $p \in add_o \Rightarrow p \notin del_o$

- $p \in pre_o \Rightarrow p \notin add_o$
- $pre_o \neq \emptyset \wedge (add_o \cup del_o) \neq \emptyset$.

Počet těchto modelů určí délku prvního vektoru. Následně naplní oba vektory náhodnými čísly.

Dále v každém kroku pro každý vzorový plán vybere náhodný model a k němu vygeneruje posloupnost akcí, které byly pravděpodobně doopravdy vykonány. Pravděpodobnost vybrání jednotlivých modelů je určena vektorem vah. Generované posloupnosti začínají počátečním stavem vzorového plánu a prováděné akce jsou vybírány náhodně s pravděpodobností danou $p(a|a_0, s, m)$, která je počítána pomocí druhého vektoru. Pro každou takovou dvojici (model, vygenerovaná posloupnost) následně vypočítá hodnotu funkce odměny hodnotící, jak kvalitní tato dvojice je. Pomocí této hodnoty upraví čísla ve vektorech a pokračuje další vstupní posloupností.

Po určitém počtu kroků se AMAN zastaví a vrátí model s největší vahou.

Podle autorů tento přístup funguje pro bezchybné plány podobně dobře jako ARMS. Stejně jako on neklade žádné zvláštní požadavky na vstupní plány. Navíc je flexibilnější, protože dokáže pracovat i s mírně chybnými vstupy.

## 3.3 Opmaker

Opmaker [7] je jeden ze starších přístupů a slouží pouze pro zjednodušení popisu světa uživatelem, neřeší problém automaticky. Umí pracovat jen se světy, ve kterých lze každý objekt popsat stavovým automatem s parametrizovanými stavy.

Na vstupu dostává seznam typů, seznam predikátů, které popisují stavy objektů jednotlivých typů a vzorové posloupnosti akcí s počátečními a koncovými stavy.

Program prochází akce v trénovací sekvenci a když narazí na nenaučenou akci, nalezne postupně pro každý její parametr všechny stavy, které můžou být parametrizovány ostatními parametry akce, a dá uživateli na výběr, na který stav akce objekt změní. Nakonec vypíše schémata naučených operátorů.

Vzhledem k tomu, že se Opmaker neučí plně automaticky, nelze ho plně zařadit k ostatním. V tomto přehledu ho uvádím hlavně kvůli jeho nástupci Opmaker2.

## 3.4 Opmaker2

Opmaker2 [6] je vylepšením Opmakeru. Oproti jeho předchůdci nepotřebuje dodatečný vstup od uživatele, místo toho ale na vstupu navíc vyžaduje seznam logických výroků, které by ve světě měly stále platit (invariantů).

Invarianty jsou definovány v jazyce predikátové logiky prvního řádu. Invariant může tedy vypadat například takto:

$$\forall H : hub.[fastened(H) \Leftrightarrow \exists N : nuts.(tight(N, H) \vee loose(N, H))]$$

Tento výrok říká, že v každém stavu musí platit, že v něm může být predikát $(fastened\ ?H)$, kde $?H$ je objekt typu $hub$, právě tehdy, pokud existuje nějaký

objekt $?N$ typu *nuts*, pro který je ve stavu predikát (*tight* $?N$ $?H$) nebo predikát (*loose* $?N$ $?H$). Jednodušeji řečeno: šroub musí být ve stavu označen jako používaný právě tehdy, když je na něm nějaká matice (utažená či povolená).

Na vstupu Opmaker2 očekává seznam typů a predikátů, které se v daném světě používají, invarianty a vzorový plán. U každého operátoru potřebuje informaci, které parametry při provedení akce mění stav a které ne.

Opmaker2 prochází sekvenci akcí a u každé, podobně jako v Opmakeru, projde všechny měnící se parametry a zjistí, jak se může změnit jejich stav. Toto provede se všemi parametry akce. Následně provede kartézský součin množin možných stavů všech proměnných a najde takovou n-tici stavů, ve které platí invarianty. Vytvoří z ní popis operátoru a pokračuje další akcí ve vstupním plánu.

Můžeme mít například model domény *Blocksworld*, ve kterém by jednotlivé kostky mohly mít 5 možných stavů:

```
S1 = [holding(o)]
S2 = [on-table(o), clear(o)]
S3 = [on-table(o), on(p, o)]
S4 = [on(o, q), clear(o)]
S5 = [on(o, q), on(p, o)].
```

Řekněme, že je akce (*unstack* $o$ $p$) provedena ve chvíli, kdy je objekt $o$ ve stavu [*on*($o$, $p$), *clear*($o$)] a objekt $p$ ve stavu [*on-table*($p$), *on*($o$, $p$)]. Programu je řečeno, že oba objekty použitím této akce mění stav, každý má 4 možnosti, jak se může změnit, celkem tedy vygeneruje 16 možností, jak může akce fungovat. Následně pro každou možnost zkontroluje, zda v ní platí invarianty. Pokud bychom například měli invariant, že pouze jedna kostka ve světě může být ve stavu S1 (tedy že jeřáb může v jednu chvíli držet pouze jednu kostku), vyloučil by program možnost, že oba objekty přejdou do stavu S1. Postupně by měl program procházením všech invariantů vyloučit všechny možnosti až na jednu, kterou použije pro definici operátoru.

Hlavní problém tohoto přístupu je vymýšlení invariantů. Vytvoření takových pravidel, které mu pomůžou najít jednoznačné řešení, může být často náročné. Tento přístup tedy uživateli usnadňuje tvorbu modelu jen zčásti, protože pouze přesouvá náročnost tvorby do jiné roviny. Další jeho nevýhodou je, že klade omezení na to, jaké domény se dokáže naučit. Musí to být modely jejichž objekty se dají popsat stavovým automatem.

## 3.5 LOCM

LOCM [4] se od předchozích přístupů odlišuje tím, že vytváří vlastní reprezentaci modelu a ignoruje původní predikátový popis. Pro každý typ objektu vytvoří konečný automat s parametrizovanými stavy, který určuje, jak objekt mění svůj stav, když vykonává jednotlivé akce.

LOCM pracuje ve dvou fázích. V první fázi buduje konečné automaty. Pro každý typ projde všechny operátory, které mohou dostat jako parametr objekt daného typu, a pro každou pozici v seznamu parametrů, na kterou může být tento objekt umístěn, vytvoří dva stavy: stav před provedením akce a stav po ní. Následně pro každý objekt ve světě vybere ze vstupních sekvencí akcí ty akce,

které ho používají. Postupně tyto nové posloupnosti prochází a pro každou po sobě jdoucí dvojici akcí unifikuje odpovídající stavy.

Mějme například plán domény *Blocksworld*, ve kterém jeřáb staví věže z kostek:

```
(unstack b4 b3)
(putdown b4)
(unstack b2 b1)
(stack b2 b3)
(pickup b1)
(stack b1 b2)
(pickup b4)
(stack b4 b1)
```

LOCM ho projde a pro objekt $b2$ v něm najde akce (*unstack* $b2$ $b1$), (*stack* $b2$ $b10$) a (*stack* $b1$ $b2$). Následně unifikuje stavy $unstack.1.after$ a $stack.1.before$ a stavy $stack.1.after$ a $stack.2.before$. Toto provede pro všechny objekty, které jsou v plánu použity. Po projití všech plánů vytvoří pro každý typ ze zbylých stavů konečný automat.

V druhé fázi se snaží zjistit, které stavy vytvořených automatů by mohly mít parametry. Pro každý stav vezme všechny dvojice operátorů, které mohou být použity pro vstup a výstup ze stavu a projde jejich seznamy parametrů. Pro dvojice pozic, do kterých může být předán stejný objekt, vytvoří hypotézy, že tyto parametry akcí jsou i parametry daného stavu. Následně opět pro každý objekt projde vstupní sekvence a ověřuje platnost těchto hypotéz. Nevyvrácené hypotézy týkající se stejných parametrů postupně spojuje a z těch, které ukazují na nějaký parametr každého vstupního i výstupního operátoru daného stavu, nakonec vytvoří parametry stavů.

Hlavní výhodou tohoto přístupu oproti ARMS nebo AMAN je, že naučené modely popisují vstupní plány bez chyb. Je to ale vykoupené tím, že vytváří vlastní reprezentaci světa, která je navíc často příliš zjednodušená. Vezměme si například jednoduchý svět, ve kterém auto převáží náklad z místa na místo pomocí 3 operátorů: (*move* ?*truck* ?$from$ ?*to*), (*load* ?*truck*, ?*place* ?*package*) a (*unload* ?*truck* ?*place* ?*package*). LOCM si všimne, že operátor *move* bývá ve vstupních plánech použit jak po operátoru *load*, tak po operátoru *unload*. Unifikuje tedy stavy po provedení těchto operátorů a nakonec dostane konečný automat s jediným stavem, který má parametr typu *place*. Kdybychom přitom místo jednoho operátoru *move* použili dva operátory ($move - loaded$ ?*truck* ?$from$ ?*to* ?*package*) a ($move - empty$ ?*truck* ?$from$ ?*to*) vytvořil by korektní automat s 2 stavy, z nichž jeden by měl navíc parametr pro balíček, který je momentálně v nákladním autě (pokud bychom tedy nepovolili převážení více balíčků najednou).

Zajímavou vlastností tohoto přístupu je, že se neučí dobře z opravdových plánů, které se snaží najít cílové predikáty v co nejkratším čase, ale pracuje lépe se vstupy, které jsou tvořeny náhodnými korektními posloupnostmi akcí. LOCM předpokládá, že každá dvojice akcí, které lze s jedním objektem za sebou provést, se ve vstupních datech objeví. Pokud se v nich nějaká dvojice neobjeví, většinou vytvoří pro daný typ automat, ve kterém tato dvojice akcí nemůže být provedena. Ve skutečných plánech se některé korektní páry akcí nikdy neobjeví, například proto, že druhá akce vrací svět do stavu před provedením první akce, a tedy daný

pár nedává z pohledu plánovače smysl. LOCM z takovýchto plánů může vytvořit horší automat, než z posloupností, ve kterých jsou přítomny všechny korektní páry akcí.

Vezměme si například opět model domény *Blocksworld*. Pokud za sebou provedeme akce (*stack* $b1$ $b2$) a (*unstack* $b1$ $b2$), svět bude opět ve stavu jako před provedením těchto akcí. V žádném opravdovém plánu se tedy tyto akce za sebou neobjeví. Pokud předáme LOCM na vstupu takové plány, vytvoří kvůli tomuto chybějícímu páru automat se čtyřmi stavy bez parametrů. Pokud bychom ale do vstupních dat přidali plán s tímto párem akcí, vytvoří přesnější konečný automat se třemi stavy: kostka je ve vzduchu, kostka je na zemi a nic na ní není a kostka je na zemi a je na ní kostka určená parametrem.

## 3.6 LOCM2

Přístup LOCM2 [5] je vylepšením svého předchůdce, LOCM. Snaží se řešit problém, že automaty, které LOCM vytvoří, jsou často příliš zjednodušené.

Mějme například opět model domény *Blocksworld*, ve kterém jeřáb staví věže z kostek. Budeme používat 4 operátory: (*pickup* ?*block*), (*putdown* ?*block*), (*stack* ?*block* ?*underblock*) a (*unstack* ?*block* ?*underblock*). LOCM vytvoří konečný automat se 3 stavy popisující vršek kostky: buď je kostka ve vzduchu, nebo je položená a nic na ní není a nebo je položená a je na ní jiná kostka. Ale tento automat už nedokáže popsat, zda kostka stojí na zemi nebo na jiné kostce, protože příliš přísně unifikuje stavy. LOCM2 toto řeší tím, že pro některé typy vytvoří více konečných automatů, které dohromady popisují stav objektu z různých pohledů. Ne všechny musí mít definovány přechody pro všechny možné operátory (pokud automat narazí na akci, pro který nemá definovaný přechod, tak stojí). V tomto případě LOCM2 vytvoří ještě druhý automat popisující, co je pod kostkou pomocí 3 stavů: kostka je ve vzduchu, kostka je na zemi nebo je kostka na jiné kostce.

LOCM2, podobně jako jeho předchůdce, pro daný typ objektů vytvoří seznam přechodů, tedy dvojic typu (název operátoru, číslo parametru), kde je daný typ používán. Následně ze vstupních dat vytvoří přechodovou matici reprezentující, které přechody se vyskytují po kterých. V této matici pak hledá takové čtveřice přechodů (r, r', c, c'), kde $(r, c) \notin M, (r, c') \in M, (r', c) \in M, (r', c') \in M, r \neq r'$ a $c \neq c'$, kde M je přechodová matice a $r$, $c$, $r'$ a $c'$ jsou indexy nějakých přechodů. (r,c) pak označí jako díru. Díra je místo v M, které LOCM automaticky zaplňoval, kvůli čemuž byly výsledné automaty příliš obecné.

Mějme například přechodovou matici pro předchozí příklad:

| | 1. | 2. | 3. | 4. | 5. | 6. |
|---|---|---|---|---|---|---|
| 1. pickup.1 | | x | x | | | |
| 2. putdown.1 | x | | | o | x | |
| 3. stack.1 | o | | | x | x | |
| 4. unstack.1 | | x | x | | | |
| 5. stack.2 | | | | | | x |
| 6. unstack.2 | x | | | x | x | |

Tabulka 3.1: Přechodová matice pro model domény blocksworld.

Písmenem $x$ jsou znázorněny pozorované dvojice přechodů a písmeno $o$ reprezentuje díru v matici. Například písmeno $x$ na pozici (1,2) znamená, že ve vstupních plánech byla s nějakým objektem $b$ provedena akce (*pickup* $b$) a následně akce (*putdown* $b$). Mezera na pozici (1,1) znamená, že s žádným objektem nebyla dvakrát za sebou provedena akce (*pickup* $?o$).

LOCM díry v matici automaticky vyplňuje, což např. pro díru (1,3) znamená, že kostka ve výsledném modelu může být položena na jinou kostku, a hned poté zvednuta ze stolu.

LOCM2 postupně prochází seznam děr a každou se snaží vysvětlit tím, že najde nejmenší možnou množinu přechodů S (pokud taková existuje), indukující přechodovou matici M[S] takovou, že:

- $r, c \in S$
- M[S] neobsahuje díry
- je možné ověřit korektnost M[S] na vstupních datech.

V našem případě obě díry vysvětlí podmnožinou 1, 2, 3, 4, která indukuje následující matici:

| | 1. | 2. | 3. | 4. |
|---|---|---|---|---|
| 1. pickup.1 | | x | x | |
| 2. putdown.1 | x | | | |
| 3. stack.1 | | | | x |
| 4. unstack.1 | | x | x | |

Tabulka 3.2: Matice vysvětlující díry v matici modelu domény blocksworld.

Automat vytvořený z této matice již korektně popisuje, co se děje se spodní částí kostky, tedy že buď je ve vzduchu, na stole nebo na jiné kostce.

V poslední fázi LOCM2 projde vzniklé množiny, odstraní takové, které jsou podmnožinou jiných, přidá množinu všech přechodů a nakonec pro každou zbylou množinu vytvoří konečný automat podobně jako v LOCM. Díky tomu, že žádná M[S] neobsahuje díry, nehrozí už, že by se v tomto kroku ztratila nějaká informace.

Přístup má velmi podobné vlastnosti jako jeho předchůdce. Také je vhodný pouze pro určitý typ modelů a také se učí lépe z náhodných posloupností akcí než z opravdových plánů. Liší se pouze v tom, že v mnoha případech tvoří specifičtější a tedy lepší modely.

## 3.7 Shrnutí

Ze zmíněných přístupů je jednoznačně nejflexibilnější AMAN. Neklade žádné zvláštní požadavky na vlastnosti učených modelů a navíc dokáže pracovat i s mírně zašuměným vstupem. ARMS vyžaduje přesné vstupy, ale taktéž dokáže pracovat s libovolným modelem. Nevýhodou těchto přístupů ale je, že negarantují přesné řešení.

V tomto ohledu pracují lépe ostatní přístupy. LOCM, LOCM2 i Opmaker2 generují modely, které vysvětlí vstupní plány bez chyb. V případě LOCM a LOCM2 je to ale vykoupeno tím, že vytvářejí vlastní reprezentaci světa a Opmaker2 vyžaduje dodatečné informace o doméně v podobě invariantů. Navíc všechny tyto přístupy fungují dobře jen pro modely, jejichž objekty jdou popsat stavovým automatem. Jsou to tedy pouze situační postupy, které nelze používat v obecném případě. Pro správná zadání ale fungují dobře, LOCM2 obzvláště.

Opmaker je v zásadě pouze poloautomatický nástroj, nelze ho plně zařadit mezi ostatní přístupy.

Vlastnosti zmíněných přístupů můžeme na závěr shrnout následující tabulkou:

| | Reprezentace světa | Dodatečné informace | Výsledný model je bez chyb | Řeší chyby ve vstupních plánech |
|---|---|---|---|---|
| ARMS | predikáty | ne | ne | ne |
| AMAN | predikáty | ne | ne | ano |
| Opmaker | KA | ano | ano | ne |
| Opmaker2 | KA | ano | ano | ne |
| LOCM | KA | ne | ano | ne |
| LOCM2 | skupina KA | ne | ano | ne |

Tabulka 3.3: Porovnání učících se algoritmů.

# 4. Přístup LOUGA

V této kapitole si představíme nový přístup LOUGA (Learning operators using genetic algoritms), využívající k učení genetické algoritmy.

Přístup pracuje v několika krocích. Nejprve za použití definic predikátů a částečných definic operátorů vstupního modelu spočítá potřebnou délku genomu jedince. Každý jedinec bude kódovat, které predikáty jsou v add a del seznamech operátorů vytvářeného modelu. V druhém kroku průchodem vstupních plánů odstraní možné hodnoty některých genů, aby zmenšil prohledávaný prostor. Třetím krokem je samotná evoluce. Může probíhat dvěma způsoby: buď se generuje celý model najednou, nebo se evoluce pouští opakovaně pro každý typ predikátu zvlášť. V posledním kroku LOUGA opětovným průchodem vstupními plány dogeneruje precondition seznamy operátorů.

## 4.1 Vstupní data

LOUGA na vstupu očekává částečný popis modelu bez add, del a precondition seznamů operátorů a vstupní plány. Každý plán by měl mít kompletní počáteční stav a ideálně i kompletní koncový stav. Kompletní koncové stavy nejsou nutné k funkci programu, ale bez nich problém mívá typicky více řešení. Pokud problém nemá jednoznačné řešení, najde LOUGA libovolný model, který vysvětluje vstupní plány bez chyb, ale nemusí to být nutně původní model.

Vstupní posloupnosti můžou být libovolné korektní posloupnosti akcí. LOUGA nepředpokládá, že jsou to skutečné plány hledající cílový stav v co nejkratším počtu kroků jako například ARMS.

Korektní v tomto případě znamená i to, že by neměly nastávat situace, kdy akce přidává do stavu predikát, který už v něm je, nebo maže predikát, který ve stavu není. LOUGA bere takové akce jako chybné a nezvládne kvůli nim najít původní model.

## 4.2 Příprava evoluce

LOUGA v prvním kroku nejprve pro každý operátor vygeneruje seznam všech možných predikátů, které operátor může do světa přidat. Jinak řečeno: nalezne všechny dvojice (predikát, operátor) takové, že parametry predikátu jsou odkazy na parametry operátoru správných typů. Každá taková dvojice bude odpovídat jednomu genu. Program si určí jednoznačné pořadí těchto dvojicí a získá tak bijekci mezi množinou možných jedinců a množinou všech možných modelů, ve kterých chybí precondition seznamy operátorů.

Mějme například model s těmito predikáty a operátory:

```
(:predicates
       (at ?thing - (either object briefcase) ?l - place)
       (empty ?b - briefcase)
       (free ?thing - (either object briefcase))
       (in ?thing - object ?b - briefcase))
```

```
(:action mov0
        :parameters (?o - object ?m - place ?l - place))
(:action mov1
        :parameters (?o - briefcase ?m - place ?l - place))
(:action put-in
        :parameters (?x - object ?l - place ?b - briefcase))
(:action take-out
        :parameters (?x - object ?l - place ?b - briefcase))
```

LOUGA vygeneruje v tomto pořadí tyto dvojice:

```
 1. ((at ?o ?m), (mov0 ?o ?m ?l))
 2. ((at ?o ?l), (mov0 ?o ?m ?l))
 3. ((free ?o),  (mov0 ?o ?m ?l))
 4. ((at ?o ?m), (mov1 ?o ?m ?l))
 5. ((at ?o ?l), (mov1 ?o ?m ?l))
 6. ((empty ?o), (mov1 ?o ?m ?l))
 7. ((free ?o),  (mov1 ?o ?m ?l))
 8. ((at ?x ?l), (put-in ?x ?l ?b))
 9. ((at ?b ?l), (put-in ?x ?l ?b))
10. ((empty ?b), (put-in ?x ?l ?b))
11. ((free ?x),  (put-in ?x ?l ?b))
12. ((free ?b),  (put-in ?x ?l ?b))
13. ((in ?x ?b), (put-in ?x ?l ?b))
14. ((at ?x ?l), (take-out ?x ?l ?b))
15. ((at ?b ?l), (take-out ?x ?l ?b))
16. ((empty ?b), (take-out ?x ?l ?b))
17. ((free ?x),  (take-out ?x ?l ?b))
18. ((free ?b),  (take-out ?x ?l ?b))
19. ((in ?x ?b), (take-out ?x ?l ?b))
```

Evoluce tedy bude probíhat na jedincích s genomem délky 19.

Jednotlivé geny můžou v průběhu evoluce nabývat 3 hodnot

- 0: operátor predikát do světa nepřidává, ani ho z něj nemaže
- 1: operátor predikát přidává do světa
- 2: operátor predikát maže ze světa.

Jedinec s genem 12200 00000 00000 0000 tedy bude odpovídat modelu, kde operátor $(mov0\ ?o\ ?m\ ?l)$ bude do světa přidávat predikát $(at\ ?o\ ?m)$ a mazat ze světa predikáty $(at\ ?o\ ?l)$ a $(free\ ?o)$ a zbývající operátory nebudou dělat nic. Originální model, tedy model s takto definovanými operátory:

```
(:action mov0
        :parameters (?o - object ?m - place ?l - place)
        :effect (and (at ?o ?l) (not (at ?o ?m))))
(:action mov1
        :parameters (?o - briefcase ?m - place ?l - place)
```

```
        :effect (and (at ?o ?l) (not (at ?o ?m))))
(:action put-in
        :parameters (?x - object ?l - place ?b - briefcase)
        :effect (and (in ?x ?b) (not (empty ?b))
                                (not (free ?x)) (not (at ?x ?l))))
(:action take-out
        :parameters (?x - object ?l - place ?b - briefcase)
        :effect (and (not (in ?x ?b)) (empty ?b)
                                (free ?x) (at ?x ?l)))
```

bude odpovídat jedinci s genomem 21021 00202 20110 1102.

Genetický algoritmus záměrně negeneruje precondition seznamy. To proto, že je jednodušší vygenerovat je dodatečně, když už máme hotové add a del seznamy, než je generovat všechny najednou (viz kapitola 4.5). Evoluce takto prohledává mnohem menší prostor možných genů a díky tomu postupuje mnohem rychleji.

V druhém přípravném kroku se LOUGA snaží co nejvíce zredukovat počet možností, jak může genom vypadat. Prochází vstupními plány a při průchodu každým z nich si udržuje 2 seznamy: seznam predikátů, které jsou určitě v daném stavu, a seznam predikátů, které ve stavu být můžou, ale nemusí. Na začátku každého plánu si do prvního seznamu dá všechny predikáty z počátečního stavu a poté prochází plánem. Pro každou akci si vygeneruje všechny možné predikáty, které může do světa přidat. Pokud je predikát určitě ve světě, LOUGA si uloží, že ho daná akce nemůže do světa přidávat. Pokud predikát určitě nemůže být v daném stavu, akce ho určitě nemůže smazat. Následně všechny vygenerované predikáty přesune do druhého seznamu a pokračuje další akcí. Když narazí na stav, dá si opět všechny jeho predikáty do prvního seznamu. Pseudokód této části je uveden v tabulce Algoritmus 2.

Mějme například následující plán:

```
(:state
        (empty b1)
        (at b1 home)
        (free b1)
        (free pencil)
        (at pencil home)
        (at rubber home)
        (free rubber))
(put-in pencil home b1)
(mov1 b1 home office)
```

Víme, že ve stavu před první akcí je přítomno 7 vypsaných predikátů a žádný jiný. Akce (*put-in pencil home* $b1$) může pracovat s predikáty (*at pencil home*), (*at* $b1$ *home*), (*empty* $b1$), ($free$ *pencil*), ($free$ $b1$) a (*in pencil* $b1$). Páry tvořené těmito predikáty a operátorem *put-in* odpovídají genům 8-13. Predikáty (*at pencil home*), (*at* $b1$ *home*), (*empty* $b1$), ($free$ *pencil*) a ($free$ $b1$) jsou určitě přítomné ve stavu před provedením akce, akce je tedy nemůže do světa znova přidat. V praxi to znamená, že v průběhu evoluce nemůžou geny 8-12 nabývat hodnoty 1. Ve stavu chybí chybí predikát (*in pencil* $b1$), což znamená, že jej akce nemůže smazat, gen 13 tedy nemůže nabývat hodnoty 2.

**Algoritmus 2** Vyřazení možných hodnot některých genů průchodem vstupního plánu.

**Vstup:** plán P; pole M, reprezentující možné hodnoty genů
**Výstup:** upravené pole M

```
1: Q ← predikáty z počátečního stavu
2: R - prázdná množina predikátů
3: for all akce a z P do
4:     vygeneruj množinu predikátů X, které může a do stavu přidat nebo z něj odebrat
5:     for all p ∈ X do
6:         if p ∈ Q then                          ▷ p určitě je v aktuálním stavu
7:             M[(a,p),add] = false
8:             R = R ∪ {p}
9:             Q = Q \ {p}
10:        else if p ∉ R then                     ▷ p určitě není v aktuálním stavu
11:            M[(a,p),del] = false
12:            R = R ∪ {p}
13:        end if
14:        if po a je rozpoznaný stav S then
15:            Q = Q ∪ S
16:            R = R \ S
17:        end if
18:    end for
19: end for
```

Následně přesuneme všechny tyto predikáty do druhého seznamu a smažeme je z prvního, pokud v něm byly přítomny. Po zpracování akce máme tedy následující seznamy:

```
1. seznam
        (at rubber home)
        (free rubber)
2. seznam
        (empty b1)
        (at b1 home)
        (free b1)
        (free pencil)
        (at pencil home)
        (in pencil b1)
```

Další akcí je akce $(mov1\ b1\ home\ office)$. Operátoru $mov1$ odpovídají geny 4-7. Akce může tedy pracovat s predikáty $(at\ b1\ home)$, $(at\ b1\ office)$, $(empty\ b1)$ a $(free\ b1)$. Predikát $(at\ b1\ office)$ není ani v jednom seznamu, akce jej tedy nemůže smazat a gen 5 nemůže nabývat hodnoty 2. Ostatní predikáty jsou již v druhém seznamu, geny 4, 6 a 7 tedy zůstanou nezměněny. Program následně přidá predikát $(at\ b1\ office)$ do druhého seznamu a pokračuje další akcí.

Po zpracování všech vstupních plánů probíhá samotná evoluce.

## 4.3 Fitness funkce

Genetický algoritmus používá jednoduchou funkci počítající chybovost modelu reprezentovaného jedincem. Funkce počítá počet provedených operací a všímá si 3 druhů chyb:

- add chyba - akce se snaží přidat do stavu predikát, který už v něm je,
- del chyba - akce se snaží ze stavu smazat predikát, který v něm není,
- observation chyba - ve stavu chybí predikát, který byl ve vstupním plánu, nebo přebývá predikát, který ve stavu být neměl.

Formálně můžeme tyto chyby definovat takto: nechť $S$ je stav plánu provedeného podle modelu reprezentovaného jedincem, $T$ je množina predikátů, které byly ve vstupním plánu pozorovány v odpovídajícím stavu, $a$ je akce provedená ze stavu $S$ a $p \in add_a$, $q \in del_a$, $s \in S$ a $t \in T$ jsou predikáty. Add chyba nastává ve chvíli, když $p \in S$, del chyba nastává, když $q \notin S$ a observation chyba nastává, když $t \notin S$. Pokud je množina $T$ označena jako úplný stav, pak je za observation chybu počítáno i když $s \notin T$.

Po projití všech plánů vrátí funkce hodnotu spočítanou pomocí následujícího vzorce:

$$(1 - (error_{add} + error_{del})/(total_{add} + total_{del}) * (1 - error_{obs}/total_{obs} * c),$$

kde $error_{add}$, $error_{del}$ a $error_{obs}$ jsou počty odpovídajících chyb, $total_{add}$ a $total_{del}$ je celkový počet provedených add a del operací, $total_{obs}$ je počet zkontrolovaných predikátů ve stavech a $c$ konstanta z intervalu [0,1] definovaná uživatelem.

Při vymýšlení fitness funkce bylo cílem vytvořit takový vzorec, který dává hodnoty v rozsahu [0,1] a platí pro něj, že čím menší poměr chyb k celkovému počtu operací či pozorovaných predikátů, tím vyšší fitness. Díky první vlastnosti dokážeme na první pohled poznat, kdy jsme narazili na perfektního jedince, a druhá vlastnost je nezbytná pro řízení směru evoluce genetickým algoritmem.

Zpočátku jsme pracovali s jednodušší verzí

$$(1 - (error_{add} + error_{del} + error_{obs})/(total_{add} + total_{del} + total_{obs}),$$

ale ukázalo se, že v některých případech nefunguje dobře. V plánech bývá často akcemi provedeno mnohonásobně více add a del operací, než kolik je predikátů v mezistavech. Oddělení observation chyb od add a del chyb ve fitness pomáhá, aby se pro takové vstupní data evoluce nevydala směrem, který vytváří pěkné posloupnosti akcí, ale ignoruje rozpoznané predikáty. Současně pokud je ve vstupních posloupnostech málo rozpoznaných predikátů, je možné zmenšit hodnotu konstanty $c$, aby malé množství observation chyb až příliš neřídilo směřování populace (ale stále ho dostatečně ovlivňovalo).

Zároveň nám ale nepřišlo dobré oddělovat *add* a *del* operace, protože spolu úzce souvisí, navzájem se značně ovlivňují a bývá jich řádově podobný počet. Nicméně i vzorec

$$(1 - error_{add}/total_{add}) * (1 - error_{del}/total_{del}) * (1 - error_{obs}/total_{obs} * c)$$

by fungoval podobně dobře.

Takto definovaná fitness tedy dobře řídí směr vývoje populace, protože upřednostňuje všestranné jedince s malým množstvím chyb. Také nám umožňuje jednoduše odlišit jedince bez chyb od ostatních, protože takoví jedinci budou mít fitness hodnotu 1.

## 4.4 Operátory genetického algoritmu

Algoritmus pro evoluci používá 4 operátory.

Prvním je standardní mutace. V každé generaci má určitou malou šanci mutovat jednotlivé jedince v populaci. Když je jedinec mutován, má každý gen malou šanci změnit hodnotu, pokud je to možné (ve 2. přípravném kroku mohly být vyloučeny pro daný gen všechny možnosti až na jednu).

Druhým operátorem je operátor křížení. Přístup používá standardní jednobodové křížení. Zkoušeli jsme i jiné možné verze, které například seskupí geny podle operátorů, které reprezentují, a noví jedinci jsou tvořeni tím způsobem, že pro každý operátor vezmou odpovídající geny náhodně buď z otce nebo matky, nicméně efekt na výkon přístupu byl neznatelný, proto jsme nakonec zůstali u standardní verze křížení.

Tento operátor genetického algoritmu funguje tak, že v každé generaci náhodně páruje jedince v populaci a zkříží je. To probíhá tím způsobem, že je vygenerován náhodný bod v genomech rodičů a vytvoří se 2 noví jedinci tak, že první jedinec si vezme první část genomu od matky a druhou od otce a druhý jedinec si vezme zbývající části. Všichni noví jedinci jsou přidáni do populace k jejich rodičům a jsou odebráni nejhorší, aby zůstal pouze definovaný počet jedinců.

Třetí operátor je operátor lokálního prohledávání. Spustí se pouze ve chvíli, kdy se populace přestane vyvíjet a delší dobu stagnuje. Vyzkouší všechny možnosti, jak lze změnit jeden gen v genomu nejlepšího jedince tak, aby vznikl lepší jedinec. Z nalezených jedinců a jejich rodičů vybere daný počet nejkvalitnějších a z těch vytvoří novou populaci.

Poslední operátor vychází z [8]. Pokud populace nadále stagnuje, restartuje populaci a uloží si nejlepšího jedince do seznamu staré populace. Příště ve chvíli, kdy se populace opět dostane do stejné situace, zkusí před restartem zkřížit aktuálně nejlepšího jedince s jedinci ze staré populace. Snaží se tak využít informace z předchozích nalezených lokálních maxim k nalezení globálního maxima. Pokud za sebou proběhne větší množství restartů bez toho, že by se stará populace jakkoli zlepšila, algoritmus jí smaže a začne úplně od začátku.

## 4.5 Dodatečné generování precondition seznamů

Před tím, než program zobrazí uživateli model, který jedinec reprezentuje, musí ještě pro každý operátor vygenerovat jeho precondition seznam.

Uživatel má na výběr, jestli chce generovat negativní podmínky nebo ne. Tyto podmínky vyžadují, aby daný predikát nebyl ve světě před provedením akce. Jsou to tedy podmínky typu:

```
(not (at ?r ?l))
```

První část podmínek si program vygeneruje z add a del seznamů operátorů. Pokud operátor maže predikát ze světa, měl by daný predikát být přítomen ve stavu před akcí. Analogicky pokud operátor do světa predikát přidává, neměl by daný predikát být ve stavu před provedením akce.

Zbytek podmínek program vytvoří dodatečným průchodem vstupních plánů. Pro každý operátor si ukládá, kolikrát byl daný predikát před provedením odpovídající akce přítomen a kolikrát chyběl. Pokud byl predikát nepřítomen maximálně v uživatelem definovaném procentu případů, program ho přidá do precondition seznamu daného operátoru. Analogicky pokud nebyl až na malý počet výjimek přítomen, vytvoří z něj negativní podmínku operátoru. Pseudokód tohoto procesu si můžeme prohlédnout v sekci Algoritmus 3.

## 4.6 Generování modelu predikát po predikátu

Pro složité[1] modely mají vytvoření jedinci příliš mnoho genů a nalezení dobrého řešení trvá dlouho. Program proto podporuje ještě druhou verzi běhu, kdy se učí operátory zvlášť pro každý typ predikátu.

V praxi tento přístup znamená, že se genetický algoritmus pustí zvlášť pro každý predikát. V jednotlivých instancích fitness funkce ignoruje všechny predikáty kromě predikátů daného typu a geny popisující jiné predikáty mají povoleny pouze hodnotu 0. Genetické algoritmy běží se střídají po jedné generaci, program

[1] Za složitý model považujeme model, který obsahuje hodně typů predikátů a operátorů. Takové modely mívají dlouhé genomy, takže genetický algoritmus musí prohledat velký stavový prostor.

**Algoritmus 3** Dodatečné generování precondition seznamů

**Vstup:** genom G; množina vstupních plánů Q; maximální velikost chyby C
**Výstup:** model M

```
1: vytvoř model M reprezentovaný G
2: Y, N - pole čísel indexované stejně jako G; iniciované 0
3: for all P ∈ Q do
4:     s ← počáteční stav P
5:     for all a ∈ P do
6:         for all predikát p, který může být generován a do
7:             g ← index genu odpovídající (p,a)
8:             if p ∈ s then
9:                 Y[g]++
10:            else
11:                N[g]++
12:            end if
13:        end for
14:        s ← s po aplikování a dle M
15:    end for
16: end for
17: for all gen g ∈ G; g odpovídá predikátu p a operátoru o do
18:    if G[g] = 0 then
19:        if N[g]/(N[g] + Y[g]) < C then
20:            přidej p do pre_o
21:        else if Y[g]/(N[g] + Y[g]) <= C then
22:            přidej (not p) do pre_o
23:        end if
24:    else if G[g] = 1 then
25:        přidej (not p) do pre_o
26:    else if G[g] = 2 then
27:        přidej p do pre_o
28:    end if
29: end for
```

tak může průběžně zobrazovat dobrá řešení a nehrozí, že by příliš dlouho hledal řešení pro jeden predikát a kompletní model by nikdy nenašel.

Tento mód generuje stejný genom, jenom se ho učí po částech. Tyto části jsou na sobě nezávislé, neboť to, jestli je predikát jednoho typu přítomen ve stavu nebo ne, nemůže ovlivnit, zda je přítomnost predikátu jiného typu chybná nebo ne. Toto rozdělení problému na části je tedy korektní a dává stejné řešení, jako původní postup.

# 5. Experimentální výsledky

V této kapitole experimentálně ověříme výkon přístupu LOUGA. Porovnáme ho s existujícím přístupem ARMS a vyzkoušíme, jak si počíná v různých situacích.

## 5.1 Metodika měření

Pro každý test jsme pomocí přiloženého generátoru vygenerovali 200 náhodných plánů. Při generování dat pro experimenty porovnávající přístupy ARMS a LOUGA jsme data generovali módem „BFS and choose random goals“ s přiměřeným minimálním počtem akcí vzhledem k danému modelu. Ostatní data jsme generovali pomocí možnosti „náhodné akce“. V každém plánu jsme vygenerovali 10-20 akcí a nechali jsme vypsat kompletní vstupní a koncové stavy.

Pro samotné testování jsme používali metodu křížové validace. Náhodně jsme rozdělili vygenerovaná data na 5 skupin po 40 plánech a při každém testu jsme používali 4 skupiny jako vstup programu a pátou jako testovací množinu. Při každém testování jsme provedli 5 testů, pokaždé s jinou testovací skupinou, a zprůměrovali výsledky. (Je to stejná metodika, kterou používali autoři ARMS při testování svého přístupu.)

Ve většině testů budeme porovnávat chybovosti jednotlivých operací, jak byly popsány v kapitole 4.3 o fitness funkci.

Experimenty prováděny na notebooku s procesorem Intel Core i5-2410 2.3GHz a 8GB operační paměti.

ARMS byl pouštěn s následujícím nastavením: váha operátorů: 0.8 - 1, váha informačních klauzulí: 0.8, probability treshold 0.05, GSAT, 30 pokusů, restart treshold 60, random choice probability 0.5. (Bližší vysvětlení viz příloha - uživatelská dokumentace)

LOUGA byl většinou pouštěn s nastavením: úplný cílový stav, mód predikát po predikátu, kompletní precondition seznamy, population size: 10, maximum generací: 100000, treshold for population restart: 15, treshold for old population restart: 15, treshold for local search: 7, treshold for crossover: 10, observation error weight: 1, maximal error rate for predicates: 0. Jak se v jednotlivých případech nastavení lišilo je popsáno v odpovídajících kapitolách.

Ve výsledných tabulkách většinou měříme čas běhu a chybovosti jednotlivých operací. Chyby definujeme stejně jako v kapitole 4.3 o fitness funkci přístupu LOUGA. Add chyba tedy vzniká ve chvíli, když se akce snaží do světa přidat predikát, který už v něm je. Analogicky definujeme i del chybu. Pre chyba vzniká, když ve světě před provedením akce chybí predikát, který akce požaduje, nebo v něm je predikát, který akce zakazuje. Obs chyba vzniká, když ve stavu plánu provedeném podle výsledného modelu chybí predikát, který v něm byl v plánu odpozorován. Add ER (Error rate) a Del ER, pak definujeme jako poměr chybných operací k celkovému počtu těchto operací v plánech, Pre ER jako poměr nesplněných podmínek k celkovému počtu zkontrolovaných podmínek a Obs ER jako poměr chybějících predikátů k celkovému počtu predikátů rozpoznaných v mezilehlých a koncových stavech plánů.

## 5.2 Přehled domén

Experimenty jsme prováděli na modelech pěti různých domén. Můžeme je rozřadit do tří kategorií podle obtížnosti, s jakou se je lze naučit. Nejjednodušší jsou modely domén *Briefcase*, kterou jsme používali v kapitole 4.2, a *Blocksworld*. Středně těžká doména je *Flattyre* a těžké jsou *Rover* a *Freecell*.

Tyto domény jsme vybrali, protože jsou to obecně známé domény, používané v různých soutěžích či celkově pro testování plánovacích postupů. Pro experimenty jsme až na výjimky používali modely používající typy.

Základní vlastnosti používaných modelů můžeme shrnout Tabulkou 5.1.

| | Briefcase | Blocksworld | FlatTyre | Rover | Freecell |
|---|---|---|---|---|---|
| # typů objektů | 3 | 1 | 5 | 7 | 3 |
| # typů predikátů | 4 | 5 | 12 | 25 | 11 |
| # operátorů | 3 | 4 | 13 | 9 | 10 |
| Průměrná velikost effect seznamů | 3.3 | 4.5 | 2 | 3 | 5.6 |
| Průměrný počet parametrů operátorů | 2.66 | 1.5 | 2.33 | 4 | 4.9 |

Tabulka 5.1: Porovnání modelů používaných v experimentech.

## 5.3 Implementace ARMS

Tato práce si dala za cíl porovnat nově vytvořený přístup s již existujícím přístupem řešící víceméně stejné zadání. Budeme ho tedy porovnávat s ARMS. Bohužel v článku, který ho popisuje, chybí některé technické detaily. Pokoušel jsem se kontaktovat autory přístupu, ale bezúspěšně, takže přiložená implementace metody nepracuje tak dobře, jako verze používaná v originálním článku. V této části popíšeme nejasnosti, které mohly potenciálně zhoršit výkon používané implementace.

V první řadě v článku nebylo přesně popsáno, jak přesně se přiřazují váhy jednotlivým akcím, a tak je program přiřazuje náhodně v uživatelem daném rozmezí.

Druhý problém je, že článek přesně nevysvětluje, jak se vybírá, který predikát vysvětlí výskyt častého páru akcí. V článku byl pouze příklad v podobě: „pokud je tento predikát vybrán, pak musí platit jedna z následujících možností“, ale nebylo dál rozvedeno, který predikát to přesně bude. Přiložená implementace to tedy řeší po svém. Místo standardních klauzulí vytvoří pro každý častý pár „multiklauzuli“, tedy výrok v disjunktivně normální formě říkající, že jeden predikát musí být pro vysvětlení páru vybrán, a pokud je vybrán, musí o něm najednou platit několik výroků, jak bylo popáno v kapitole 3.1.

Program tedy nemůže používat standardní MAX-SAT algoritmus, ale musí pracovat s vlastní implementací algoritmů GSAT či WalkSAT. To se bohužel projevilo i na výkonu, a tak přiložený program nezvládá v rozumném čase pracovat se složitějšími doménami.

## 5.4 Porovnání s ARMS

V této sekci budeme přímo porovnávat výsledky přístupů LOUGA a ARMS. Experimenty rozdělíme na 4 části.

V první části budeme používat plány s úplnými cílovými stavy, cílovými predikáty a malým množstvím predikátů rozpoznaných v mezistavech (každý predikát má šanci 5%, že bude vypsán). Tedy vstup, na který byl vytvářen ARMS. V druhé části použijeme plány s úplnými počátečními a koncovými stavy bez mezistavů. Tedy vstup, který nejvíce vyhovuje přístupu LOUGA. V třetí části budeme vygenerujeme 10 různých vstupů s proměnlivým množstvím predikátů v mezilehlých a koncových stavech a budeme sledovat, jak se množství predikátů projeví na výkonu přístupů. Nakonec porovnáme přesnost přístupu LOUGA s výsledky z originálního článku popisujícího ARMS, ve kterém byly pro testování použity i modely domén Rover a Freecell.

Jelikož přiložená implementace ARMS nedosahuje kvalit a rychlosti originálu, zaměříme se v prvních dvou částech pouze na 3 lehčí modely. Ve třetí části budeme používat pouze model domény *Blocksworld*.

Pro první test má LOUGA vypnuté nastavení „cílový predikát je úplný“. Ostatní nastavení odpovídají popisu v kapitole 5.1

| | Add ER | Del ER | Pre ER | Obs. ER | Čas běhu [s] |
|---|---|---|---|---|---|
| Briefcase - ARMS | 0.263 | – | 0.029 | 0 | 6.197 |
| Blocksworld - ARMS | 0.409 | 0.095 | 0.039 | 0.001 | 28.825 |
| Flat tyre - ARMS | 0.319 | 0.479 | 0.342 | 0.003 | 504.195 |
| Briefcase - LOUGA | 0 | 0 | 0 | 0 | 0.515 |
| Blocksworld - LOUGA | 0 | 0 | 0 | 0 | 2.245 |
| Flat tyre - LOUGA | 0 | 0 | 0 | 0 | 4.187 |

Tabulka 5.2: Porovnání přístupů ARMS a LOUGA na vstupech s cílovými predikáty a malým množstvím predikátů v mezistavech.

| | Add ER | Del ER | Pre ER | Obs. ER | Čas běhu [s] |
|---|---|---|---|---|---|
| Briefcase - ARMS | 0.318 | – | 0.032 | 0 | 6.728 |
| Blocksworld - ARMS | 0.331 | 0.061 | 0.036 | 0.014 | 29.645 |
| Flat tyre - ARMS | 0.336 | 0.507 | 0.311 | 0.005 | 548.09 |
| Briefcase - LOUGA | 0 | 0 | 0 | 0 | 0.337 |
| Blocksworld - LOUGA | 0 | 0 | 0 | 0 | 1.278 |
| Flat tyre - LOUGA | 0 | 0 | 0 | 0 | 2.743 |

Tabulka 5.3: Porovnání přístupů ARMS a LOUGA na vstupech s kompletními koncovými stavy.

Ze získaných dat je vidět, že použitá implementace ARMS má problém s generováním del seznamů operátorů. Ve velké části případů nevygenerovala v celém modelu žádnou delete operaci.

Problém s generováním del seznamů měl i LOUGA na vstupech s částečnými koncovými stavy (tabulka 5.2), i když se neprojevil zdaleka tak markantně. Výsledné modely jsou sice bez jediné chyby, nicméně to nejsou přesně původní modely, některé delete operace chybí, protože pro vysvětlení vstupních dat nebyly potřeba. Může to být částečně i špatně vygenerovanými vstupy, nicméně obecně se dá říci, že mu chyběla informace, zda je potřeba nějaké predikáty mazat nebo ne. Pokud na vstupu dostane plány s úplnými koncovými stavy, tuto informaci tím získá a při dostatečném vstupu nachází typicky modely identické s původními.

Z tabulek 5.2 a 5.3 je také vidět, že je rychlost přístupu ARMS výrazně svázanější s komplexností učeného modelu. Nicméně to je zčásti způsobené i použitou implementací.

V následujícím experimentu se zaměříme pouze na doménu *Blocksworld* a budeme používat několik různých vstupů s proměnlivým množstvím predikátů v mezilehlých a koncových stavech. Každý predikát v těchto stavech se ve vygenerovaných datech objeví s pravděpodobností danou hodnotou v levém sloupci následujících tabulek.

| % predikátů | Add ER | Del ER | Pre ER | Obs. ER | Čas běhu [s] |
|---|---|---|---|---|---|
| 0 | 0.299 | – | 0.38 | 0.291 | 4.501 |
| 5 | 0.268 | 0.872 | 0.369 | 0.109 | 13.397 |
| 10 | 0.166 | 0 | 0.225 | 0.033 | 15.726 |
| 15 | 0.238 | 0 | 0.224 | 0.041 | 13.557 |
| 20 | 0.331 | 0.025 | 0.066 | 0.011 | 18.33 |
| 30 | 0.401 | 0.145 | 0.035 | 0.005 | 17.606 |
| 40 | 0.376 | 0.075 | 0.039 | 0.005 | 18.493 |
| 60 | 0.374 | 0.081 | 0.027 | 0.002 | 21.342 |
| 80 | 0.383 | 0 | 0.006 | 0 | 21.363 |
| 100 | 0.333 | 0 | 0.007 | 0 | 22.623 |

Tabulka 5.4: Výkon přístupu ARMS na vstupech domény *Blocksworld* s proměnlivým množstvím predikátů ve stavech.

| % predikátů | Add ER | Del ER | Pre ER | Obs. ER | Čas běhu [s] |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0.647 |
| 5 | 0 | 0 | 0 | 0 | 0.863 |
| 10 | 0 | 0 | 0 | 0 | 1.197 |
| 15 | 0 | 0 | 0 | 0 | 1.217 |
| 20 | 0 | 0 | 0 | 0 | 1.455 |
| 30 | 0 | 0 | 0 | 0 | 1.707 |
| 40 | 0 | 0 | 0 | 0 | 1.772 |
| 60 | 0 | 0 | 0 | 0 | 2.017 |
| 80 | 0 | 0 | 0 | 0 | 2.165 |
| 100 | 0 | 0 | 0 | 0 | 2.128 |

Tabulka 5.5: Výkon přístupu LOUGA na vstupech domény *Blocksworld* s proměnlivým množstvím predikátů ve stavech.

Z výsledků v tabulce 5.4 je patrné, že více informací ve vstupních datech ARMS jednoznačně prospívá. I v tomto testu se ale projevilo, že má problém s mazáním predikátů: operací delete bylo provedeno výrazně méně než operací add. Taktéž je vidět, že při dostatečném vstupu operace add vygenerovaly všechny potřebné predikáty, kvůli absencím některých predikátů v del seznamech operátorů ale operace add často selhávaly a to se projevilo na chybovosti této operace, která se výrazněji nezmenšila ani s přibývajícím množstvím informace ve stavech.

Oproti tomu v tabulce 5.5 je vidět, že přístup LOUGA pokaždé našel řešení, které vysvětlovalo vstupní data bez chyb. Často to nebylo původní řešení, nicméně z pohledu přístupu je nemožné nalézt původní tvar modelu, pokud jsou vstupní data nedostatečně specifická.

V poslední části budeme porovnávat výkon přístupu LOUGA s experimentálními výsledky z originálního článku ARMS. Toto porovnání je spíše orientační, protože jsme testy neprováděli za identických podmínek, nicméně i tak může být zajímavé. V rámci originálního článku bylo na daných doménách provedeno více testů s různým probability tresholdem, vybíráme ty testy, které dopadly nejlépe (všechny testy na těchto doménách ale dopadly víceméně stejně, tak na výběru použitého testu příliš nezáleží). Autoři používali pro ohodnocení výkonu primárně precondition error rate, proto budeme v této části používat pouze tu.

| | Pre ER | Čas běhu [s] |
|---|---|---|
| Rover - ARMS | 0.66 | 167 |
| Freecell - ARMS | 0.47 | 387 |
| Rover - LOUGA | 0 | 8.91 |
| Freecell - LOUGA | 0 | 7.91 |

Tabulka 5.6: Porovnání výkonu přístupu LOUGA s výsledky testů z originálního článku o ARMS.

Z tabulky 5.6 a všech ostatních částí testování je zřejmé, že přístup LOUGA pracuje na testovacích datech o poznání přesněji a zároveň rychleji než ARMS.

## 5.5 Evoluce vs. evoluce predikát po predikátu

V této části porovnáme výkon módu predikát po predikátu přístupu LOUGA s výkonem základní verze tohoto přístupu. Budeme je porovnávat na modelu domény Flat tyre. Obě verze konzistentně nachází bezchybná řešení, proto porovnáváme pouze časy běhů.

| | Čas běhu [s] | Směrodatná odchylka |
|---|---|---|
| Základní verze | 80.21 | 77.67 |
| Predikát po predikátu | 2.74 | 0.69 |

Tabulka 5.7: Porovnání výkonu základní verze přístupu LOUGA s módem predikát po predikátu.

Dle očekávání je verze predikát po predikátu o poznání rychlejší než základní verze. Co je však zajímavější, jsou směrodatné odchylky. Časy jednotlivých běhů základní verze se velmi lišily, některé byly i desetinásobně delší než jiné. Takto často fungují genetické algoritmy: rychlost nalezení perfektního řešení závisí z nemalé části na náhodě. Oproti základní verzi pracuje verze predikát po predikátu poměrně konzistentně. V rámci všech testů provedených pro tuto práci, se časy jednotlivých řešení lišily v nejhorším případě dvojnásobně.

Díky tomu, že se vytváří model pro každý predikát zvlášť, má evoluce mnohem jasnější směr, než když se generuje celý model najednou. Dá se říci, že najednou generuje pouze jednu vlastnost, i když je tvořena velkým množstvím genů. Oproti tomu standardní verze se musí zaměřovat na několik vlastností najednou, zlepšení genomu v jednom směru může přinést zhoršení v jiném. Díky tomu módu je přístup LOUGA relativně konzistentní i při složitějších zadáních (viz následující kapitola).

## 5.6 Svět bez typů

Učící se přístupy jsou většinou dělány pro světy používající typy. Pro přístupy, které objekty popisují konečnými automaty, jsou typy naprosto nezbytné, ostatním alespoň značně zmenší prohledávaný prostor. Prakticky ale typy nejsou vůbec potřeba, lze je adekvátně reprezentovat i predikáty. Je určitě zajímavé vyzkoušet, jak si LOUGA poradí se světy, které typy vůbec nepoužívají. Provedeme tedy testy na modelech domén Freecell a Rover a výsledky porovnáme s výsledky z učení se standardních verzí těchto domén (používali jsme stejná vstupní data jako při posledním experimentu porovnávání s ARMS).

Pro tuto fázi jsme vypnuli průběžné generování kompletních precondition seznamů, protože kvůli velikosti vstupních dat by tato funkce zpomalila počítání každé generace o cca 5 sekund. Tento seznam je možné dogenerovat jednorázově po skončení běh genetického algoritmu.

Výsledky měření si můžeme prohlédnout v tabulce 5.8. Vzhledem k tomu, že LOUGA ve všech případech našel bezchybné řešení, neuvádíme vůbec chybovost. Rovněž neporovnáváme vlastnosti modelů, protože se pouze lehce liší v precondition seznamech.

| | Délka genomu | Čas běhu [s] | Směrodatná odchylka |
|---|---|---|---|
| Rover s typy | 201 | 8.90 | 0.91 |
| Freecell s typy | 291 | 7.91 | 0.44 |
| Rover bez typů | 2796 | 249.64 | 51.29 |
| Freecell bez typů | 1481 | 67.4 | 10.99 |

Tabulka 5.8: Výkon přístupu LOUGA v závislosti na tom, zda model používá typy.

Z výsledků je patrné, že se odstraněním typů značně zvětšila délka tvořeného genomu a kvůli tomu i čas běhu programu. Nicméně je zřejmé, že LOUGA zvládne vyřešit i takový typ zadání.

Nové geny v genomu můžeme rozdělit na 2 skupiny: první skupinou jsou geny tvořené novými predikáty popisující typy. Tyto geny prakticky neovlivní průběh učení, protože LOUGA už na začátku zjistí, že odpovídající podproblémy mají triviální řešení (predikáty nejsou používány v žádném add nebo delete seznamu).

Druhou skupinou jsou geny odvezené od predikátů, které v modelu byly už před odstraněním typů. Tyto geny už složitost problému výrazně zvýší. Z modelu domény Rover bylo vytvořeno násobně víc takovýchto genů, protože má podobný počet operátorů a jejich parametrů, ale výrazně více typů predikátů i typů objektů. Po odstranění typů můžou všechny parametry operátoru potenciálně přidávat libovolný predikát, počty typů, predikátů, operátorů a jejich parametrů se tedy velmi projeví na složitosti výsledného problému a délce jeho řešení.

# Závěr

V prvních dvou kapitolách jsme definovali základní pojmy a struktury, se kterými jsme v této práci pracovali, a přesně vymezili problém, který řeší výslední přístup LOUGA. V další kapitole jsme probrali několik již existujících přístupů řešících různé druhy podobných problémů a rozebrali, jaké jsou jejich silné stránky a kde naopak mírně zaostávají.

Dále jsme představili nový přístup LOUGA a podrobně probrali, jakým způsobem funguje. Na závěr jsme experimentálně ověřili výkonnost navrženého přístupu LOUGA a ukázali, že pro testovaná data dokáže pracovat rychleji a výrazně přesněji než existující přístup ARMS. Také jsme ukázali, že zvládne pracovat i s relativně náročnými zadáními v podobě světů nepoužívající typy.

V rámci práce vznikly dvě aplikace. Hlavní z nich, pojmenovaná Model Learner, umožňuje zobrazovat, upravovat a testovat na chyby vstupní data ve variantě jazyka PDDL. Dále umožňuje z těchto dat generovat modely za použití čtyř různých přístupů: ARMS, LOCM, LOCM2 a LOUGA. Všechny přístupy v průběhu běhu zobrazují základní informace o svém stavu, uživatel tedy může na konkrétních datech analyzovat, jakým způsobem fungují. Druhá aplikace je pojmenovaná Plan Generator a slouží pouze pro generování vstupních dat pro učící se přístupy.

V budoucnu by přístup LOUGA mohl být rozšířen, aby nějakým způsobem dokázal pracovat i se složitějšími konstrukcemi jazyka PDDL, například aby podporoval i číselné hodnoty a jednoduché aritmetické výrazy. Zajímavou úpravou by mohlo být i upuštění od požadavku na kompletní vstupní stav. Jednoduššími rozšířeními by mohla být podpora podmínky na ekvivalenci v precondition seznamech operátorů nebo podpora dědičnosti typů objektů jak v rámci aplikace, tak i v rámci funkčnosti přístupu LOUGA. Samotná aplikace by také mohla být v další práci rozšířena, aby podporovala přidávání nových přístupů za běhu v podobě pluginů.

Další částí této práce jsou přílohy, ve kterých nalezneme uživatelskou a programátorskou dokumentaci jak pro hlavní program, tak i pro generátor plánů.

# Seznam použité literatury


[1] Melanie Mitchell. *An Introduction to Genetic Algorithms.* MIT Press, Cambridge, MA, USA, 1998.

[2] Qiang Yang, Kangheng Wu, and Yunfei Jiang. Learning action models from plan examples using weighted MAX-SAT. *Artif. Intell.*, 171(2-3):107–143, 2007.

[3] Hankz Hankui Zhuo and Subbarao Kambhampati. Action-model acquisition from noisy plan traces. In *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*, pages 2444–2450, 2013.

[4] Stephen Cresswell, Thomas Leo McCluskey, and Margaret Mary West. Acquisition of object-centred domain models from planning examples. In *Proceedings of the 19th International Conference on Automated Planning and Scheduling, ICAPS 2009, Thessaloniki, Greece, September 19-23, 2009*, 2009.

[5] Stephen Cresswell and Peter Gregory. Generalised domain model acquisition from action traces. In *Proceedings of the 21st International Conference on Automated Planning and Scheduling, ICAPS 2011, Freiburg, Germany June 11-16, 2011*, 2011.

[6] Thomas Leo McCluskey, Stephen Cresswell, N. Elisabeth Richardson, and Margaret Mary West. Action knowledge acquisition with opmaker2. In *Agents and Artificial Intelligence - International Conference, ICAART 2009, Porto, Portugal, January 19-21, 2009. Revised Selected Papers*, pages 137–150, 2009.

[7] Thomas Leo McCluskey, N. Elisabeth Richardson, and Ron M. Simpson. An interactive method for inducing operator descriptions. In *Proceedings of the Sixth International Conference on Artificial Intelligence Planning Systems, April 23-27, 2002, Toulouse, France*, pages 121–130, 2002.

[8] Deniz Yuret. From genetic algorithms to efficient optimization. Technical report, Cambridge, MA, USA, 1994.

[9] Drew Mcdermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. PDDL - The Planning Domain Definition Language. Technical report, CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, 1998.

[10] Pavel Torgashov. Fast colored textbox for syntax highlighting. `https://www.codeproject.com/Articles/161871/Fast-Colored-TextBox-for-syntax-highlighting`, 2011–2014.

# Přílohy

# A. Uživatelská dokumentace

V této části se zaměříme na použití hlavní aplikace z pohledu uživatele. Představíme si GUI, projdeme možná nastavení jednotlivých přístupů a vysvětlíme si, co znamenají jejich výpisy.

## A.1 Přehled UI

Základní rozložení UI si můžeme prohlédnout na obrázku A.1

Část GUI označená číslem 1 slouží k základním operacím se soubory.

Druhá část slouží k zobrazování vstupních dat. Program umí zvýrazňovat syntaxi jazyka PDDL a zobrazovat syntaktické chyby. Nicméně primárně slouží k práci s vygenerovanými daty, proto nepodporuje další funkce, které by usnadňovaly ruční vytváření vstupů pro přístupy.

Dalšími částmi jsou seznam záložek na pravé straně a jejich obsah. První záložka slouží k volbě a nastavení přístupů (viz níže). Ostatní záložky slouží k zobrazování průběhu a výsledku učícího procesu. Pro přístupy ARMS, LOCM a LOCM2 jsou to pouze záložky „Output“ a „Solution“, pro LOUGA lze navíc prohlížet i nejlepší nalezené jedince. V případě módu „Predikát po predikátu“ je možné prohlížet stav každé instance genetického algoritmu pro jednotlivé predikáty.

Mezi jednotlivými instancemi lze přepínat pomocí seznamu označeného číslem 5. Kromě názvu predikátů je v tomto seznamu i zobrazena fitness nejlepšího dosud nalezeného jedince. Pokud byl nalezen perfektní jedinec, je predikát označen slovem „done“.

V části 6 je vidět počet chyb ve vstupních datech. Kliknutím na nápis je možné rychle přejít na následující chybu.

## A.2 Práce s jednotlivými přístupy

V následující části si představíme možnosti nastavení jednotlivých přístupů a vysvětlíme si, co znamenají jejich výpisy.

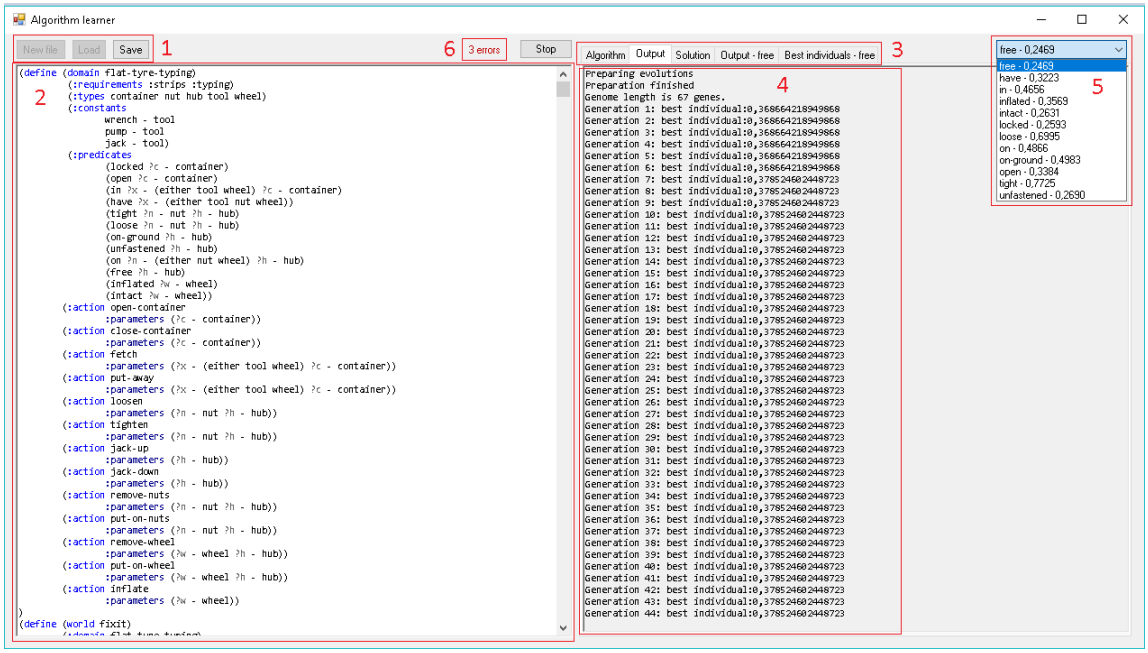
Po spuštění přístupů LOUGA nebo ARMS se část jejich UI pro nastavení uzamkne. Ty ovládací prvky, které zůstanou odemčené, je možné upravovat i za běhu přístupů, efekt některých se projeví dokonce i pro dokončení učícího procesu. Je možné takto například dodatečně dogenerovat precondition seznamy modelu, který vytvoří LOUGA.

### A.2.1 LOUGA

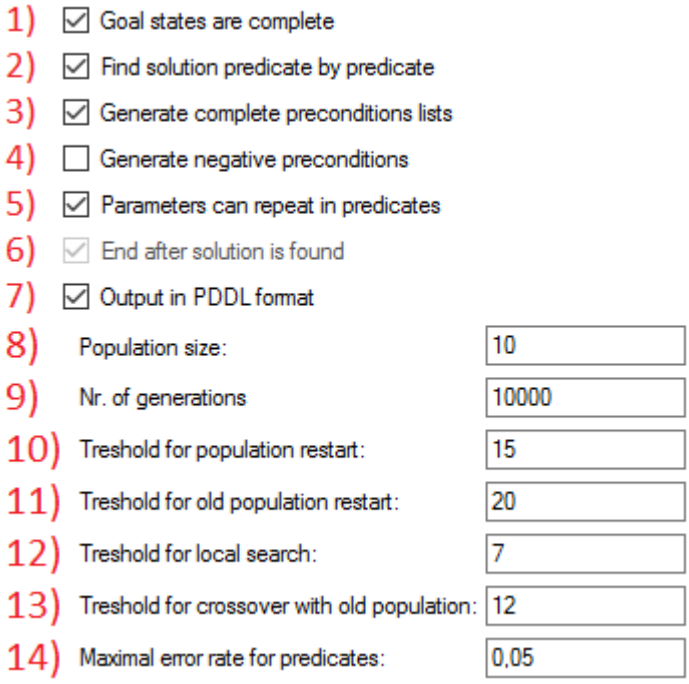
**Nastavení**

Možnosti nastavení přístupu LOUGA si můžeme prohlédnout na obrázku A.2.

První políčko umožňuje uživateli říct programu, že jsou cílové stavy ve vstupních plánech úplné. Program pak bude modely hodnotit i podle toho, jestli jsou v cílových stavech nějaké predikáty navíc nebo ne.




Obrázek A.1: GUI hlavního programu.




Obrázek A.2: Nastavení přístupu LOUGA.

Druhé políčko nastavuje přístup, aby pracoval v režimu predikát po predikátu, jak je popsáno v kapitole 4.6.

Další dvě políčka upravují, jak program generuje precondition seznamy. První zapíná dodatečné generování predikátů, jak bylo popsáno v kapitole 4.5 a druhé povoluje generování negativních podmínek.

Páté políčko určuje, zda má LOUGA v přípravném kroku generovat i predikáty, které mají více stejných parametrů. Příkladem takového predikátu může být například (*on* ?*ob* ?*ob*) pro operátor (*stack* ?*ob* ?*underob*). Takové predikáty většinou nejsou v modelu potřeba a vypnutí této funkce může znatelně zredukovat prohledávaný prostor a tím zrychlit běh programu.

Šesté políčko nastavuje, zda se má program zastavit po prvním nalezeném řešení nebo se jich snažit najít co nejvíce. Toto nastavení nelze změnit, pokud je program v režimu predikát po predikátu.

Poslední políčko nastavuje, zda program bude modely vypisovat v plném PDDL formátu, nebo zda bude vypisovat přehlednější výpis sestávající se pouze z *add* a *del* seznamů jednotlivých operátorů.

Následuje šest kolonek upravující vlastnosti běhu genetického algoritmu. První dvě nastavují velikost populace a maximální počet generací. Další 4 upravují vlastnosti některých operátorů genetického algoritmu, jak jsou popsány v kapitole 4.4. Treshold je splněn ve chvíli, když je stáří nejlepšího jedince v populaci rovno dané hodnotě (v případě restartování populace musí být stáří jedince větší nebo rovno dané hodnotě).

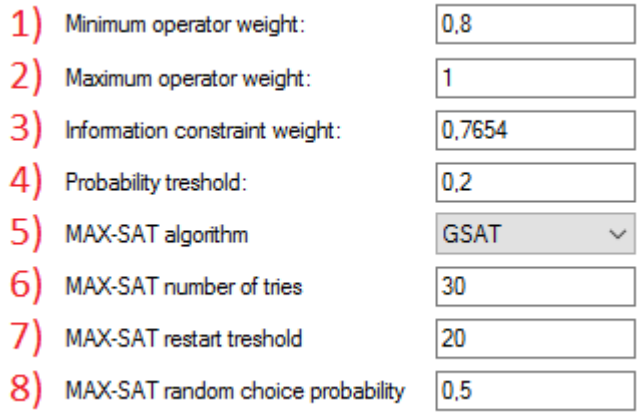
Poslední kolonka nastavuje maximální chybovost predikátů, které jsou v dodatečném kroku přidávány do precondition seznamů operátorů.

**Vysvětlení výpisů**

Výpis standardního módu běhu přístupu LOUGA může vypadat například takto:

```
Generation 27: best individual:0,970365843156044
local search
Generation 28: best individual:0,970365843156044
Generation 29: best individual:0,970365843156044
Generation 30: best individual:0,970365843156044
crossover with old population
Generation 31: best individual:0,970365843156044
Generation 32: best individual:0,970365843156044
Generation 33: best individual:0,970365843156044
Generation 34: best individual:0,970365843156044
Generation 35: best individual:0,970365843156044
------------------------------------------------------------------------
Best individual with fitness 0,970365843156044.
Former old populations:
Restarted population. Currently in old population:
0): 0,97037 - 220111121200020010210021020
------------------------------------------------------------------------
Generation 36: best individual:0,872344089471189
```

Po každé generaci je uživateli vypsáno číslo generace a fitness nejkvalitnějšího jedince. Pokud populace splňuje podmínku pro lokální prohledávání nebo




Obrázek A.3: Nastavení přístupu ARMS.

křížení se starou populací, je uživateli vypsán odpovídající výpis. Tyto operace bývají početně náročnější než běžná simulace jedné generace, programu tedy můžou chvíli trvat. Ve chvíli, kdy populace splní podmínku pro restartování, vypíše se informace o restartování a seznam jedinců, kteří jsou aktuálně ve staré populaci, případně seznam zavrhnutých starých populací. Tito jedinci se (stejně jako v záložce *Best individuals*) vypisují ve formátu fitness - genom a jsou řazení sestupně podle fitness. Pokud genetický algoritmus najde perfektního jedince, vypíše odpovídající model.

V případě módu predikát po predikátu musí LOUGA nejprve připravit prostředí pro jednotlivé genetické algoritmy. To může chvíli trvat, program tedy do hlavní záložky vypisuje průběh příprav zprávami typu:

```
Prepared evolution for predicate 'in'.
```

Po dokončení přípravy se spustí genetické algoritmy a podobně jako ve standardním módu se vypisuje číslo generace a fitness nejlepšího jedince složeného z částí vygenerovaných jednotlivými algoritmy. V případě, že některá z instancí genetického algoritmu nalezne perfektního jedince, vypíše se adekvátní zpráva do hlavního okna. Po ukončení všech instancí se vypíše informace o chybovosti nalezeného modelu (první číslo je poměr chybových operací ku celkovému počtu a v závorce jsou uvedeny počty chybných operací a celkový počet provedených operací odpovídajících typů) a délce běhu programu.

### A.2.2 ARMS

**Nastavení**

Možnosti nastavení přístupu ARMS si můžeme prohlédnout na obrázku A.3.

První dvě kolonky nastavují váhy operátorů. Každému operátoru je na začátku každé iterace učícího procesu přiřazena náhodná hodnota z daného rozsahu. Tato hodnota určuje váhu klauzulí prvního typu, které popisují formální požadavky na *add*, *del* a *precondition* seznamy operátorů a také určuje, která akce bude vybrána jako první jako naučená.

Třetí kolonka určuje váhu klauzulí druhého typu, které vysvětlují výskyty predikátů v mezistavech.

Hodnota ve čtvrté kolonce stanoví minimální podíl plánů, ve kterých se musí objevit pár akcí nebo dvojice predikát-akce, aby byly považovány za dostatečně časté pro vytvoření klauzulí třetího a čtvrtého typu.

Zbývající kolonky nastavují vlastnosti MAX-SAT algoritmu, který hledá řešení vygenerované formule. Uživatel má na výběr z algoritmů GSAT a WalkSAT. Může nastavit, kolik má algoritmus pokusů na nalezení globálního maxima, jak dlouho se může snažit dostat z lokálního maxima, než se restartuje, a s jakou pravděpodobností má vykonat náhodný krok.

**Vysvětlení výpisů**

ARMS jeden po druhém zpracovává všechny operátory. Pro každý nejprve vytvoří formuli pro MAX-SAT algoritmus a vypíše ji na výstup. Vypsaná formule může vypadat například takto:

```
Created formula:
0,850896 (!(at ?o ?m)->mov.add | !(at ?o ?m)->mov.pre)
 & 0,850896 (!(at ?o ?m)->mov.del | (at ?o ?m)->mov.pre)
 & 0,850896 (!(at ?o ?l)->mov.add | !(at ?o ?l)->mov.pre)
 & 0,800000 (at ?b ?l)->take-out.add
 & 0,800000 !(at ?o ?l)->mov.del
 & 0,800000 (at ?o ?l)->mov.add
 & 0,362694 (((at ?o ?l)->mov.del) |
                ((free ?o)->mov.del) |
                ((empty ?o)->mov.pre & !(empty ?o)->mov.del) |
                ((empty ?o)->mov.add))
Max Weight: 5,315382
```

První číslo na řádku určuje váhu následující klauzule (popř multiklauzule). Pro disjunkci je používán znak |, pro negaci vykřičník a pro konjunkci znak &. Každá klauzule je vypsána vždy na jeden řádek, multiklauzule je pro přehlednost vypsána na více řádků tak, že každý řádek je konjunkcí literálů. Literál $(at\ ?o\ ?m)$->$mov.add$ odpovídá výroku, že predikát $(at\ ?o\ ?m)$ je v $add$ seznamu operátoru $mov$. Na konci výpisu formule je vypsán součet vah jejích klauzulí a multiklauzulí.

Formule je následně vyřešena MAX-SAT algoritmem. Použitý algoritmus má na vyřešení několik pokusů, na konci každého vypíše váhu nejlepší nalezeného řešení. Po provedení všech pokusů se vybere nejlepší nalezené řešení a vypíše se naučený tvar operátoru.

Po vyřešení všech operátorů se podobně jako u přístupu LOUGA vypíšou informace o chybovostech jednotlivých operací a celkový čas běhu programu.

### A.2.3 LOCM

Přístup LOCM nemá žádné možnosti nastavení, probereme tedy pouze, co znamená jeho výpis.

**Vysvětlení výpisů**

LOCM pracuje v několika fázích. V první prochází pro každý typ seznamy parametrů všech operátorů a hledá, na která místa je možné umístit objekt daného typu. Pro každé takové místo vypíše info v podobě operátor-index parametru (indexy začínají nulou).

V dalším kroku vytváří konečné automaty. Pro každý typ vypíše pouze počet stavů odpovídajícího automatu.

Dále tvoří hypotézy o parametrech z popisů operátorů. Výpisy o vytvořených hypotézách můžou vypadat například takto:

```
adding hypothesis: make-0-AfterAction->finish-1-BeforeAction 1 -> 0
adding hypothesis: make-0-AfterAction->start-1-BeforeAction 1 -> 0
```

První řádek oznamuje, že byla vytvořena hypotéza tvrdící, že pokaždé, když je s objektem $o$ provedena akce $(make\ o\ ?p)$ a následně akce $(finish\ ?q\ o)$, tak $?p$ a $?q$ je jeden a ten samý objekt. Prostřední část výpisu tedy popisuje, kterých operátorů se hypotéza týká a na kterých indexech je používán objekt, který mění stav. Čísla na konci řádku říkají, na kterých indexech by se měl vyskytovat parametr stavu, o kterým se hypotéza tvoří.

V dalším kroku LOCM ověřuje vytvořené hypotézy na vstupních datech. Pokud je nějaká vyvrácena, je vypsána zpráva v podobném formátu jako při vytvoření hypotézy.

Po ověření hypotéz se testuje, pro které stavy zbylo dostatek hypotéz pro vytvoření parametrů. O každém stavu každého automatu se vypíše informace, kolik má parametrů.

Na konec se vypíšou už jen informace o nových predikátech, které popisují stavy automatů a informace o délce běhu učícího programu.

### A.2.4 LOCM2

Přístup LOCM2 nemá žádné možnosti nastavení, probereme tedy pouze, co znamená jeho výpis.

**Vysvětlení výpisů**

LOCM2 nejprve průchodem vstupních dat vytváří matice. Každá vytvořená matice je vypsána v podobném formátu, jaký byl používán v kapitole 3.6. Za každou maticí je vypsán seznam nalezených děr a informace o utvořených množinách přechodů. Vypsaná matice může vypadat například takto:

```
pickup.0  . x x . . .
putdown.0 x . . x . .
stack.0   . . . x x .
stack.1   . . . . . x
unstack.0 . x x . . .
unstack.1 x . . x x .
```

Ve výpisu se v tomto případě neobjevují informace o hypotézách, protože jich LOCM2 tvoří příliš mnoho a zbytečně by zahlcovaly výpis. Zbytek výpisu je prakticky stejný jako výpis přístupu LOCM.

# B. Syntaxe používané verze jazyka PDDL

Program pro vstup a výstup používá lehce upravený jazyk PDDL [9]. PDDL je jazyk, který je standardně používán pro zápis plánovacích modelů a problémů pro plánovače. Jako takový nebývá používán pro zápis světů a plánů, a tak bylo potřeba si ho lehce upravit, aby v něm bylo možné psát vstupy pro učící se přístupy. PDDL je jinak definován velmi robustně, aby bylo možné v něm zapsat širokou škálu problémů, my budeme používat pouze jeho základní funkce.

## B.1 Definice modelu

Modely se v programu zapisují pomocí standardního PDDL. Model zapsaný v tomto jazyce může vypadat například takto:

```
(define (domain trucks)
  (:requirements :typing)
  (:types truck package location)
  (:constants home - location)
  (:predicates
      (at ?thing - (either truck package) ?location - location)
      (in ?package - package ?truck - truck)
      (empty ?truck - truck)
  )

  (:action move-loaded
      :parameters (?truck - truck ?from ?to - location ?package - package)
      :precondition (and (at ?truck ?from) (in ?package ?truck))
      :effect (and (not (at ?truck ?from)) (at ?truck ?to)))
  (:action move-empty
      :parameters (?truck - truck ?from ?to - location)
      :precondition (and (at ?truck ?from) )
      :effect (and (not (at ?truck ?from)) (at ?truck ?to))
  )
  (:action load
      :parameters (?truck - truck ?place - location ?package - package)
      :precondition (and (at ?truck ?place) (at ?package ?place)
                                            (empty ?truck))
      :effect (and (not (at ?package ?place)) (not (empty ?truck))
                                  (in ?package ?truck))
  )
  (:action unload
      :parameters (?truck - truck ?place - location ?package - package)
      :precondition (and (at ?truck ?place) (in ?package ?truck))
      :effect (and (not (in ?package ?truck)) (at ?package ?place)
                                  (empty ?truck))
  )
)
```

Klíčové slovo *define* vždy uvozuje definici modelu nebo problému. Slovo *domain* v první závorce specifikuje, že se jedná o definici doménového modelu, za ním následuje jeho název. V další závorce je za slovem :*requirements* deklarováno, které všechny syntaktické konstrukce jsou použity pro zápis modelu. Tento program pracuje pouze s nejzákladnější reprezentací domén, a tak rozumí pouze klíčovému slovu :*typing*. To říká, že model podporuje typování objektů, každý objekt pak musí mít deklarovaný svůj typ. Pokud toto klíčové slovo není definováno, všechny objekty musí být deklarovány bez typu.

Následuje seznam typů objektů ve světě a po něm deklarace konstant. Konstanty jsou objekty, které jsou přítomny ve všech světech odvozených z tohoto modelu. Pokud model podporuje typy, je potřeba pro každý objekt uvést i jeho typ. Ten se zapisuje tak, že se za název objekt napíše pomlčka a po ní název typu. Je možné takto deklarovat typ více objektů najednou.

V další části jsou definice typů predikátů. Každá z nich se skládá z unikátního jména predikátu a ze seznamu parametrů. Identifikátory parametrů vždy začínají znakem ?. Typy parametrů se definují téměř stejně jako typy konstant, jediný rozdíl je, že pokud má parametr na výběr z více typů, zapisuje se to pomocí klíčového slova *either*, za kterým je seznam možných typů.

Poslední částí jsou definice operátorů. Každá z nich začíná klíčovým slovem *action*, za kterým je unikátní název operátoru. Po něm následuje seznam parametrů operátoru, definovaný podobně jako seznam parametrů predikátu. Za ním jsou předpoklady a efekty operátoru. Ty v programu na vstupu nejsou vůbec potřeba, program pouze zkontroluje, zda jsou syntakticky správně napsány a přístupy je ignorují. Jsou zapsány pomocí logických spojek *and* a *not* a predikátů, jejichž parametry jsou odkazy na parametry operátoru.

Například akce (*move-empty truck*1 *warehouse junkyard*) vyžaduje, aby byl predikát (*at truck*1 *warehouse*) přítomen ve světě před provedením akce. Pak ho smaže a místo něj do světa přidá predikát (*at truck*1 *junkyard*).

## B.2 Definice světů a plánů

Jelikož je PDDL určen primárně pro definování problému, bylo potřeba si vytvořit vlastní syntaxi pro zápis vstupních plánů. Je ale velmi podobná standardní definici problému pro plánovače, zavádí pouze 3 nová klíčová slova. Svět odvozený z předchozího modelu může vypadat například takto:

```
(define (world truck-world)
        (:domain trucks)
        (:objects
                t1 - truck
                t2 - truck
                t3 - truck
                p1 - package
                p2 - package
                p3 - package
                l1 - location
                l2 - location
                l3 - location
                l4 - location)
```

```
        (:plan
                (:state
                        (empty t1)
                        (at t1 l1)
                        (in p1 t2)
                        (at t2 l2)
                        (empty t3)
                        (at t3 l4)
                        (at p2 l1)
                        (at p3 l4))
                (move-loaded t2 l2 l4 p1)
                (move-empty t1 l1 l3)
                (move-empty t3 l4 l1)
                (load t3 l1 p2)
                (unload t2 l4 p1)
                (:state
                        (at p1 l4)
                        (at t1 l3)
                        (in p2 t3)))
        (:plan
                (:state
                        (empty t1)
                        (at t1 l1)
                        (in p1 t2)
                        (at t2 l2)
                        (empty t3)
                        (at t3 l4)
                        (at p2 l1)
                        (at p3 l4))
                (load t1 l1 p2)
                (move-loaded t1 l1 l2 p2)
                (load t3 l4 p3)
                (unload t1 l2 p2)
                (unload t2 l2 p1)
                (:state
                        (at p1 l2)
                        (at p2 l2)
                        (in p3 t3)))
)
```

Definice světa začíná stejně jako definice modelu klíčovým slovem *define*. Za ním následuje slovo *world* říkající, že se jedná o svět, a po něm identifikátor světa. Na dalším řádku se udává název modelu platného ve světě. Všechny světy by měly odkazovat na model definovaný v tom samém souboru, světy odkazující na jiné modely jsou programem označovány jako chybné. Dále následuje seznam objektů uvozený slovem :*objects*. Objekty ve světě jsou definovány stejně jako konstanty v modelu.

Poslední částí je seznam vzorových plánů. Každý plán začíná klíčovým slovem :*plan*, za ním je posloupnost akcí a stavů. Každý stav je definován pomocí slova :*state* a množiny predikátů v něm platných. Program obecně předpokládá, že stavy nemusí být úplné, ve stavech provedených plánů tedy můžou být i predi-

káty, které nejsou vypsány. Výjimkou jsou počáteční a koncové stavy pro některé přístupy. V plánu by měly být uvedeny všechny provedené akce, neměly by se tedy nikdy objevit 2 stavy za sebou.

Ve zdrojovém souboru by měl být jediný model a alespoň jeden svět s minimálně jedním plánem.

# C. Programátorská dokumentace

V této části se zaměříme na popis implementace hlavního programu a pomocného generátoru plánů.

Aplikace byly naprogramovány v jazyce C# v prostředí Visual Studio 2010. Byla použitá jediná externí knihovna [10], implementující FastColoredTextBox, kvalitnější verzi ovládacího prvku pro zadávání textu. V průběhu vývoje se ukázalo, že standardní RichTextBox rychlostí nedostačuje potřebám vytvářené aplikace, bylo proto potřeba použít externí knihovnu.

Ve výsledku jsou zdrojové kódy rozdělené do 3 projektů. Projekty ModelLearner a PlanGenerator reprezentují jednotlivé aplikace, třetím projektem je knihovna pojmenovaná GeneralPlanningLibrary, která se skládá z tříd používaných oběma aplikacemi (například třídy sloužící k reprezentaci jednotlivých entit plánovacího prostředí či parser vstupních dat).

## C.1 Projekt ModelLearner

Hlavní třídou této aplikace je třída Form1, která obstarává veškerý vstup od uživatele a velkou část výpisu. Zbývající část výpisu zajišťuje třída OutputManager, která primárně slouží pro bezpečnou komunikaci mezi vlákny zajišťující učení a hlavním vláknem obstarávající UI aplikace.

Uživatelské nastavení aplikace se ukládá pomocí standardního .NET mechanismu v Properties.Settings souboru.

Zdrojové kódy všech přístupů jsou ve složce Algorithms, každý z nich je ve vlastní podložce. Hlavní třída každého přístupu implementuje interface ILearningAlgorithm obsahující jedinou metodu Learn(Model m, List<World> worlds).

### C.1.1 Implementace LOUGA

**Hlavní třídy**

Hlavními třídami implementace přístupu LOUGA jsou třídy LOUGA a LOUGAPredicateByPredicate. První obstarává standardní běh přístupu a druhá mód, ve kterém se učí model jeden predikát po druhém, jak je popsáno v kapitole 4.6. Způsob fungování těchto módů se z implementačního hlediska poměrně liší, bylo proto přehlednější implementovat každý z nich ve své vlastní třídě.

**Genetický algoritmus**

Další důležitou třídou je třída GeneticAlgorithm. Tato třída obstarává základní funkčnost genetického algoritmu. Pro svoje fungování vyžaduje třídu reprezentující fitness funkci a množinu policies, tedy tříd implementující interface IPolicy, které v každé generaci nějak upraví současnou populaci. Hlavní metodou třídy GeneticAlgorithm je pak metoda RunGeneration(), která postupně aplikuje jednotlivé policies na současné jedince a mezi jednotlivými voláními upravuje populaci, aby stále byla v konzistentním stavu.

**Reprezentace jedinců**

Skupina tříd v podsložce Individuals slouží k reprezentování jednotlivých jedinců v populaci. Jádrem je interface IIndividual, který definuje základní rozhraní jedince a který je implementován třídou IntegerIndividual. Třídy IndividualRepresenter a IntegerIndividualRepresenter slouží k reprezentaci mapování mezi jedinci a modely. První z nich vytváří jednoznačné pořadí dvojic (operátor, predikát) a druhá konkretizuje mapování na číselné jedince, definuje, co znamenají hodnoty jednotlivých genů, a poskytuje metodu pro převod jedince na model.

**Fitness funkce**

Rozhraní pro fitness funkce definuje interface IFitnessFunction<T> s jedinou metodou GetFitness(T individual). K samotnému počítání hodnoty fitness slouží 2 třídy IntegerErrorRateFunction a IntegerErrorRateOfSinglePredicate. Obě z nich počítají fitness způsobem popsaným v kapitole 4.3, liší se pouze v tom, že druhá se zaměřuje pouze na výskyty jednoho konkrétního typu predikátu, zatímco první počítá fitness standardním způsobem. Třída IntegerPrefixTreeWrapper slouží pouze pro zrychlení získávání hodnot fitness funkcí. Ukládá si hodnoty fitness pro již spočítané jedince a v případě, že je opakovaně dotázána na stejný genom, vrátí uloženou hodnotu. Jinak zavolá metodu GetFitness některé z předchozích tříd podle aktuálního módu běhu.

**Policy třídy**

Poslední skupinou tříd přístupu LOUGA jsou třídy, které zajišťují úpravy populace v každé generaci. Všechny tyto třídy implementují interface IPolicy <IntegerIndividual>, který vyžaduje jedinou metodu Affect(List <IntegerIndividual> population). Jmenovitě jsou to třídy SimpleMutation, OperatorSwitchCrossover, LocalSearchPolicy a IntegerRestartingPolicy, jejichž funkčnost je popsána v kapitole 4.4 o operátorech genetického algoritmu, a třída BestIndividualsLogger. Ta slouží k ukládání historicky nejlepších nalezených jedinců a jejich zobrazování uživateli pomocí výše zmíněné třídy OutputManager.

### C.1.2 Implementace ARMS

Hlavní třídou zajišťující drtivou většinu funkcionality přístupu ARMS je třída ARMS. Ostatní třídy slouží víceméně jen k reprezentaci informací a řešení vygenerované MAX-SAT úlohy.

Každá instance třídy ActionPair reprezentuje pár akcí, který se často vyskytuje v plánech společně.

K reprezentaci vygenerované formule slouží třídy Formula, Clause, Multiclause a Literal. Interface IClause sloužící pro zjednodušení práce s klauzulemi a multiklauzulemi.

Interface IMaxSatSolver definuje rozhraní pro implementace algoritmů řešící MAX-SAT problém. Je realizován třídami GSAT a WalkSat, které implementují algoritmy odpovídajících jmen, které byly lehce upraveny, aby uměly pracovat s multiklauzulemi.

### C.1.3 Implementace LOCM

Implementace LOCM a LOCM2 byly vytvořeny ještě předtím, než bylo jasné, jak přesně bude nový přístup fungovat. Vzhledem k tomu, že fungují značně odlišně než výsledná metoda a tvoří si vlastní reprezentaci světa, nebyly nakonec v experimentech použity, v aplikaci nicméně zůstaly.

Hlavní třídou implementace přístupu LOCM je třída LOCMMachine. Zajišťuje velkou část funkcionality tohoto přístupu. Při prvním průchodu vstupními daty vytváří pro každý typ objektu konečný stavový automat reprezentovaný instancí třídy PartialStateMachine. V dalším kroku každá instance PartialStateMachine vygeneruje pro každý stav množinu hypotéz o parametrech reprezentovaných instancemi třídy Hypothesis. Po jejich ověření vygeneruje pro některé stavy parametry reprezentované třídou ParameterInfo a na závěr třída LOCMMachine vygeneruje výsledný model.

### C.1.4 Implementace LOCM2

LOCM2 funguje velmi podobně jako jeho předchůdce a i je podobně strukturován. Hlavní třídou je LOCM2Machine. Třída TransitionMatrix slouží k reprezentaci stavových matic pro jednotlivé typy objektů. Zajišťuje funkcionalitu pro hledání děr a generování podmnožin přechodů pro další krok. Jednotlivé podmnožiny jsou reprezentované instancemi třídy TransitionSet, které fungují velmi podobně jako PartialStateMachine přístupu LOCM. Třídy Hypothesis a ParameterInfo jsou prakticky stejné jako odpovídající třídy v předchozí části, pouze pracují s jinými strukturami.

## C.2 Projekt PlanGenerator

Generátor plánů je naprogramován v odděleném projektu nazvaným PlanGenerator. Hlavními třídami tohoto projektu jsou třídy GeneratorMainForm, která obstarává funkčnost UI aplikace a třída Generator, která provádí samotné generování plánů způsoby popsanými v kapitole D.1 a jejich zápis do souboru. Třída PathState reprezentuje stav při BFS. Pro načítání a reprezentování dat se používají třídy z knihovny GeneralPlanningLibrary.

Uživatelské nastavení je podobně jako v hlavním projektu uloženo pomocí standardního Settings souboru.

## C.3 Knihovna GeneralPlanningLibrary

Tato knihovna obsahuje třídy používané oběma aplikacemi. Konkrétně jsou to třídy reprezentující jednotlivé entity plánovacího prostředí a třídy obstarávající parsování vstupních dat.

Všechny třídy používané pro reprezentaci plánovacích struktur jsou uloženy ve složce Structure. Konkrétně jsou to třídy Model, World, Plan, State, Predicate, PredicateInstance, PredicateReference, Operator, Action, Type a Object. Instance těchto tříd mají u sebe uloženy základní informace a odkazy na nadřazené třídy (operátor na model, objekt na jeho typ, atd...)

Třída PredicateInstance slouží k uložení konkrétních instancí predikátů ve stavech, oproti tomu třída PredicateReference je používána pro reprezentaci precondition a effect seznamů operátorů. Parametry predikátu jsou uloženy jako číselné odkazy do seznamu parametrů operátoru popřípadě odkazy na konstantní objekty modelu.

Třída Tree slouží k ukládání precondition a effect seznamů operátorů ve struktuře definované jazykem PDDL. Nicméně pro ulehčení práce si operátory interně tvoří i vlastní reprezentaci pomocí 4 seznamů predicate referencí (add, del a positive a negative preconditions).

Dekódování vstupních dat zajišťuje třída StringDomainReader. Pří dekódování nejprve provede základní lexikální analýzu vstupního souboru a převede vstup do přirozenější stromové struktury. Pro reprezentaci této struktury používá instance třídy ReaderTree. V dalším kroku se jednotlivé stromy převádějí do konečného formátu pomocí metod DecodeModel, DecodeWorld, atd... Ve všech částech dekódování se kontrolují syntaktické i sémantické chyby a předávají se volající metodě ve výstupním seznamu List<InputException> errors.

Součástí této knihovny je i složka Utility obsahující třídu Utility a pomocnou strukturu ErrorInfo. Třída Utility slouží primárně ke konečné analýze výkonnosti modelu vygenerovaného nějakým přístupem. Struktura ErrorInfo slouží k reprezentaci informací získaných při analýze o chybovosti jednotlivých operací.

# D. Dokumentace ke generátoru plánů

Pro testování přístupů je součástí práce i jednoduchý generátor náhodných plánů.

Cílem této práce nebylo vytvořit perfektní generátor, proto je tato aplikace relativně primitivní. Nepodporuje zobrazování vstupů ani výstupů a generování plánů může trvat delší dobu.

## D.1 Popis generátoru

Program ve vstupním souboru očekává popis modelu s kompletními popisy operátorů a za ním několik vzorových světů. V každém světě by měl být jeden nebo více plánů s jediným stavem, který bude počátečním stavem vygenerovaných plánů.

Program umí generovat plány třemi způsoby. První způsob pouze generuje náhodné korektní posloupnosti akcí. V druhém program opakovaně vygeneruje náhodné predikáty, které nejsou ve vstupním stavu, a snaží se najít plán, který je do světa přidá. Pro hledání používá jednoduché prohledávání do šířky. Pokud cílový stav nenalezne dostatečně rychle, začne znova s jinými predikáty. V třetím módu program z každého počátečního stavu spustí prohledávání do šířky a pro každou kombinaci až tří predikátů, které nebyly v počátečním stavu, si ukládá stav, ve kterém se tato kombinace poprvé objevila. Po určité době vybere určitý počet těchto cílových stavů a vypíše plány, které svět do těchto stavů dostaly.

Když program pracuje prvním nebo třetím způsobem, snaží se pro každý počáteční stav vygenerovat stejné množství plánů. Může se stát, že se mu nepodaří nalézt dostatečné množství plánu odpovídajících podmínkám, je pak potřeba změnit nastavení programu, či použít obecnější počáteční stavy.

Při zjišťování, zda je akce aplikovatelná na stav, program kontroluje pouze precondition seznamy. Při špatně vytvořeném modelu tedy akce může do světa přidat predikát, který už něm je, popřípadě mazat predikát, který v něm není. S takovými vstupy LOUGA neumí dobře pracovat a pravděpodobně nenalezne původní model.

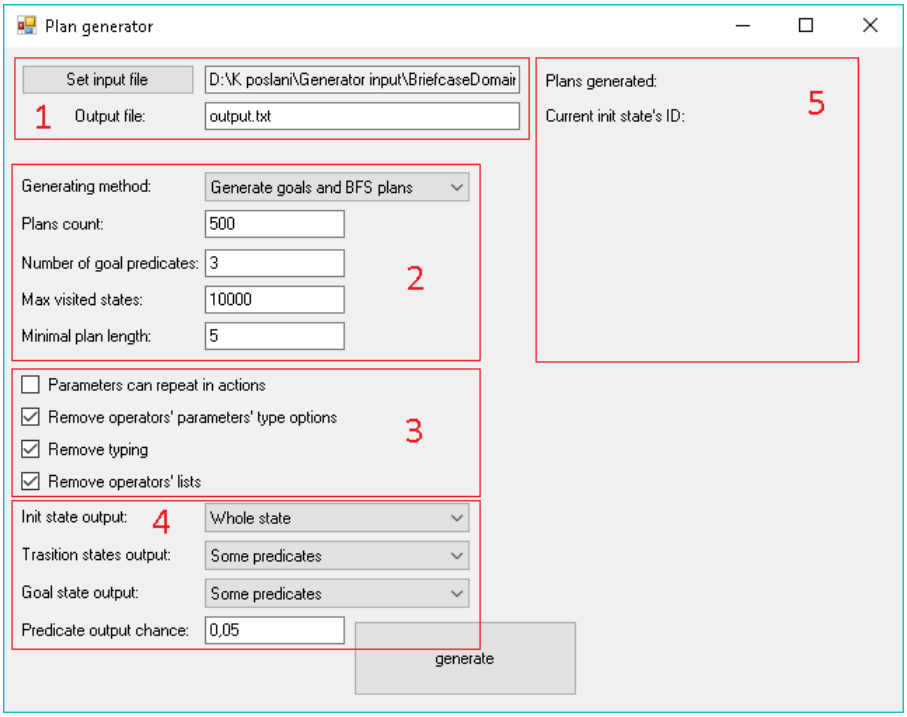
## D.2 UI generátoru

Možnosti nastavení generátoru si můžeme prohlédnout na obrázku D.1.

V první části UI lze nastavit vstupní a výstupní soubor.

V druhé části se nastavuje metoda generování, kterou bude program používat a počet plánů, který vygeneruje.

Pro první způsob generování je potřeba zadat počet akcí, jak dlouhé mají být výsledné plány. Je možné i zadat, minimální počet, jak dlouhé plány může program ještě použít, pokud se dostane do stavu, ze kterého nelze provést žádná akce.




Obrázek D.1: Možnosti nastavení generátorů plánů.

Pro druhý způsob je možné nastavit, kolik má program generovat cílových predikátů pro každý model, maximální počet prohledaných stavů, než začne program hledat od začátku, a minimální přípustnou délku vygenerovaných plánů.

Podobně pro třetí způsob je možné nastavit počet prohledaných stavů a minimální délku plánu.

V třetí části UI může uživatel zadat obecné parametry generátoru. První políčko určuje, zda jedna akce může dostat vícekrát jako parametr ten samý objekt. Další tři přepínače určují, jak se upraví vstupní model. V případě zaškrtnutí druhého políčka program najde všechny operátory, jejichž parametry mají na výběr z více typů (tedy jejich seznamy parametrů obsahují slovo *either*) a rozdělí je na několik operátorů, které mají typy parametrů přesně dané. Třetí políčko určuje, zda má program odebrat ze světa typy. V případě, že je zaškrtnuto, je pro každý typ vytvořen nový predikát a do precondition seznamu každého operátoru je pro každý jeho parametr přidán predikát odpovídajícího typu. Poslední políčko určuje pouze, zda má program vypisovat operátory bez jejich precondition a effect seznamů.

V čtvrté části UI může uživatel nastavit, kolik informace o stavech má program ve výsledných plánech uvést. Nastavení je oddělené pro vstupní stav, koncový stav a stavy mezi nimi. Pro každou skupinu je na výběr ze 3 možností: vypsat celý stav, nevypisovat nic a vypsat jen náhodné predikáty. Pravděpodobnost vybrání každého predikátu je určena hodnotou v políčku "Predicate output chance". Pro koncový stav je možné zvolit ještě možnost "Goal predicates", při které program vypíše pouze cílové predikáty vygenerovaného plánu (pokud program negeneroval pouze náhodné akce).

Poslední část UI slouží k zobrazování průběhu generování. Uživatel zde vidí počet vygenerovaných plánů, ID zpracovávaného vstupního stavu a případně stav prohledávacího algoritmu.

# E. Obsah CD

Na přiloženém CD je uložena digitální verze tohoto textu, zdrojové kódy programů, data používaná v experimentech, testovací data a spustitelné verze obou aplikací.

Popis adresářů na CD:

- Data_used_in_experiments - data použitá v experimentech
- Executables - spustitelné verze obou aplikací
- Generator_input - vstupní data pro generátor plánů
- Other_test_data - vstupní data pro testování přístupů LOCM a LOCM2
- Source - zdrojové kódy obou aplikací