POLITECHNIKA ŚLĄSKA

WYDZIAŁ AUTOMATYKI, ELEKTRONIKI I INFORMATYKI

Projekt inżynierski

Język zadań planowania PDDL

Autor: Krzysztof Palenta

Kierujący pracą: dr inż. Adam Gałuszka

Gliwice, styczeń 2011

Spis treści:

1. Język planowania zadań – PDDL

1.1. Czym jest PDDL?

Język PDDL (ang. Planning Domain Definition Language) jest próbą standaryzacji języków wykorzystywanych do opisu domen i problemów planowania[1][2][5]. Został stworzony głównie na potrzeby międzynarodowych zawodów planowania IPC (International Planning Competition) w 1998 roku. Kolejne edycje IPC zaowocowały rozszerzeniem języka PDDL o nowe funkcje i elementy.

Dużą zaletą jest obsługa popularnych systemów sztucznej inteligencji jak STRIPS (STanford Research Institute Problem Solver) czy ADL (Action Description Language). W podstawowej wersji języka wyróżnia się 3 poziomy ekspresji:

I. Planowanie ADL (ang. ADL planning) – pozwala na wyrażanie typów, własności i relacji obiektów, przypisywanie parametrów oraz wyrażanie warunków
II. Konstrukcje liczbowe (ang, numerical constructions) – wprowadza zmienne liczbowe, którym można przypisywać konkretne wartości
III. Akcje ciągłe (ang, durational actions) – umożliwiają reprezentację upływu czasu

1.2. Zadanie planowania

Zadaniem planowania nazywamy problem przekształcenia otoczenia ze stanu początkowego do docelowego przy pomocy możliwych metod jego modyfikowania[1][2]. Możemy wyróżnić 5 podstawowych składników zadania planowania:

- Obiekty (ang. objects) – przedmioty otoczenia
- Predykaty (ang. predicates) – właściwości obiektów
- Akcje/operatory (ang. actions) – metody zmiany stanu otoczenia
- Stan początkowy (ang. initial state) – sytuacja początkowa otoczenia
- Stan docelowy (ang. goal) – warunek, który chcemy by był spełniony

PDDL pozwala na kompletny opis zadania planowania. Właściwości dziedziny planowania definiujemy poprzez predykaty i akcje w części domeny, natomiast stan początkowy i docelowy określamy w części problemu. Zadanie planowania w języku PDDL składa się więc z dwóch plików: pliku domeny planowania oraz pliku problemu planowania.

1.3. Podstawy pisowni języka PDDL

Komentarze

Jak każdy język programistyczny tak i PDDL umożliwia dodawanie komentarzy do skryptów. Komentowaną linijkę poprzedza się znakiem średnika ‘;’.

```
; To jest prawidłowy komentarz
```

Zmienne

Zmienną w języku PDDL zapisujemy dodając znak zapytania ‘?’ przed jej nazwą. Przypisanie typu odbywa się za pomocą predykatów lub za pośrednictwem deklaracji TYPING (ang. typing). Aby zdefiniować nową zmienną za pomocą owej metody należy podać jej typ po znaku myślnika ‘-‘ w sposób przedstawiony poniżej.

```
; Definicja nowej zmiennej za pomoca predykatu (:strips)
TYP_ZMIENNEJ ?NAZWA_ZMIENNEJ

; Definicja nowej zmiennej (:typing)
?NAZWA_ZMIENNEJ – TYP_ZMIENNEJ
```

Typy zmiennych definiujemy w bloku *types*.

```
(:types TYP_ZMIENNEJ_1 TYP_ZMIENNEJ_2 ...)
```

Operatory

Zbiór obsługiwanych operatorów zależy od posiadanej wersji i rozszerzenia języka PDDL. Jednak w skład najbardziej podstawowych wchodzą operatory logiczne and, or i not. Ich składnia została przedstawiona w poniższym przykładzie.

```
(and (ZDANIE_LOGICZNE_1)        ; Składnia operatora „and”
     (ZDANIE_LOGICZNE_1)
...)

(or (ZDANIE_LOGICZNE_1)         ; Składnia operatora „or”
    (ZDANIE_LOGICZNE_1)
...)

(not (ZDANIE_LOGICZNE))         ; Składnia operatora „not”
```

Pisownia pozostałych operatorów jest analogiczna.

2. Definiowanie domeny

2.1. Format domeny

Definicja domeny zawiera wymagania (ang. requirements), predykaty (ang. predicates) oraz akcje i operatory (ang. actions) [2][5]:

```
(define (domain NAZWA_DOMENY)
  (:requirements [:strips] [:typing] [:adl])
  (:predicates (PREDYKAT_1 [?ARG_1 ?ARG_2 ...])
               (PREDYKAT_2 [?ARG_1 ?ARG_2 ...])
  ...)

  (:action AKCJA_1
    :parameters (?PARAM_1 ?PARAM_2 ...)
    :precondition WARUNEK
    :effect EFEKT
  )

  (:action AKCJA_2
  ...)

...)
```

2.2. Wymagania

Język PDDL zawiera w sobie wiele systemów planowania. Składnik *requirements* pozwala nałożyć wymóg obsługi konkretnych podzbiorów języka.

Tab. 1. Powszechne parametry składnika *requirements*.

Parametr	Opis
:strips	Podstawowy podzbiór PDDL, który stanowi system STRIPS
:typing	Domena używa rozszerzonej deklaracji zmiennych TYPING
:adl	Domena używa systemu ADL
:equality	Symbol '=' interpretowany jest jako operator równości

Pominięcie elementu *requirements* jest równoważne z wymogiem obsługi STRIPS.

2.3. Predykaty

Właściwości otoczenia i zależności jego obiektów definiowane są w predykatach. Przykład implementacji cechy i dwóch relacji możliwych w danej dziedzinie został zaprezentowany poniżej.

```
(:predicates (Cecha_1 ?obiekt_1)
             (Relacja_1 ?obiekt _1 ?obiekt_2)
             (Relacja_2 ?obiekt _1 ?obiekt_2 ?obiekt_3))
```

Poprzez odwołanie się do predykatów możemy identyfikować i zmieniać bieżące właściwości i relacje obiektów. W ten sposób dane o obecnym stanie otoczenia pozyskiwane są w formułach warunkowych, a ich modyfikacja i przypis jest możliwa przy deklaracji stanu początkowego oraz wykonywaniu akcji.

2.4. Akcje/operatory

Istotą planowania jest możliwość zmiany stanu danego otoczenia. Metody jego modyfikacji określa się poprzez definicję możliwych akcji w domenie zadania. Każdy blok akcji stanowią 3 elemeny:

- Parametry (ang. parameters) – obiekty potrzebne do opisu akcji
- Warunek stosowalności (ang. precondition) – stan dla którego akcja jest możliwa
- Efekt (ang. effect) – zmiana stanu otoczenia będąca konsekwencją wykonania akcji

Parametry akcji

Zmienne akcji definiujemy wymieniając je w elemencie *parameters*. Do funkcji akcji powinny zostać przekazane wszystkie obiekty opisujące warunek stosowalności oraz efekt jej wykonania.

```
:parameters (?obiekt_1 ?obiekt_2 ?obiekt_3)

; Przykładowo:
; obiekt_1, obiekt_2 – warunkują akcję
; obiekt_2, obiekt_3 – służą do reprezentacji stanu po
;                      wykonaniu akcji
```

Warunek stosowalności akcji

Warunek ten określa czy dana akcja może zostać wykonana w obecnym stanie. Formułuje się go jednym zdaniem logicznym w elemencie *precondition*.

```
; Przykład 1 (:strips)
:precondition (Cecha_1(?obiekt_1))

; Przykład 2 (:strips)
:precondition (and (Cecha_1(?obiekt_1))
                   (not (Wlasciwosc_1(?obiekt_2)))
                   (Relacja_1(?obiekt_1 ?obiekt_2)))
```

Domena używająca wymagania *equality* może zawierać warunki równości w definicji akcji.

```
; Przykład 1 (:equality)
:precondition (= (?obiekt_1) (?obiekt_2))

; Przykład 2 (:equality)
:precondition (not (= (?obiekt_1) (?obiekt_2)))
```

Formuła w domenie ADL może posiadać dodatkowo funkcje *forall* oraz *exists*.

```
; Skladnia forall (:adl)
forall (LISTA) (WARUNEK)

; Skladnia exists (:adl)
exists (LISTA) (WARUNEK)

; Przykład 1 (:adl)
:precondition (forall (Cecha_1(?obiekt_X))
                (Relacja_1(?obiekt_1 ?obiekt_X)))

; Przykład 2 (:adl)
:precondition (exists (Cecha_1(?obiekt_X))
                (Relacja_1(?obiekt_1 ?obiekt_X)))
```

Warunek z przykładu pierwszego ADL należy rozumieć jako pytanie czy wszystkie obiekty posiadające cechę „Cecha_1” są w relacji „Relacja_1” z obiektem „obiekt_1”. Analogicznie funkcja *exists* zwraca prawdę jeśli którykolwiek z obiektów o właściwości „Cecha_1” jest w takiej relacji.

Efekt akcji

Podobnie jak w przypadku warunków, efekt akcji zapisuje się w postaci jednego zdania logicznego, które ma być spełnione po jej wykonaniu. Elementy listy dopisków oraz skreśleń wypisujemy stosując operatory *and* i *not*.

```
; Przykład 1 (:strips)
:effect (not (Cecha_1(?obiekt_3)))

; Przykład 2 (:strips)
:effect (and (not (Cecha_1(?obiekt_3)))
             (not (Relacja_1(?obiekt_2 ?obiekt_3)))
             (Relacja_1(?obiekt_3 ?obiekt_2)))
```

W formule efektu nie można stosować operatora równości '='. Jest to konsekwencją niejednoznaczności jego zaprzeczenia '*(not (= ...))*'.

Używanie systemu ADL przez domenę rozszerza możliwości efektów o funkcje *when* oraz *forall*.

```
; Skladnia when (:adl)
when (WARUNEK) (EFEKT)

; Skladnia forall (:adl)
forall (LISTA) (EFEKT)

; Przykład 1 (:adl)
: effect (when (Cecha_1(?obiekt_2))
           (Relacja_1(?obiekt_2 ?obiekt_3)))

; Przykład 2 (:adl)
: effect (forall (Cecha_1(?obiekt_X))
           (Relacja_1(?obiekt_1 ?obiekt_X)))
```

Działanie funkcji *when* odpowiada strukturze IF-THEN. W przykładzie 1 relacja zostanie przypisana jedynie jeśli spełniony zostanie predykat „Cecha_1" przez obiekt „obiekt_2". Natomiast funkcja *forall* przypisuje określone relacje dla każdego obiektu z listy.

3. Definiowanie problemu

3.1. Format problemu

W pliku problemu określa się przynależność do domeny (ang. domain), obiekty (ang. objects), stan początkowy (ang. init) oraz docelowy (ang. goal) [2][5]:

```
(define (problem NAZWA_PROBLEMU)
  (:domain NAZWA_DOMENY)
  (:objects OBIEKT_1 OBIEKT_2 ...)

  (:init RELACJA_POCZATKOWA_1
         RELACJA_POCZATKOWA_2
  ...)

  (:goal STAN_DOCELOWY)
)
```

3.2. Deklaracja domeny i obiektów

Każdy problem pisany w języku PDDL musi mieć określoną domenę, której dotyczy. W pierwszej kolejności w elemencie *domain* wpisuje się więc jej nazwę.

Obiekty i zmienne definiuje się w elemencie *objects* poprzez wypisanie ich nazw.

```
; Przykład (:strips)
(:objects obiekt_1 obiekt_2 obiekt_3)
```

Jeśli domena problemu używa deklaracji typów *typing*, to podczas definicji przypisujemy zmiennym typy w sposób przedstawiony poniżej.

```
; Przykład 1 (:typing)
(:objects obiekt_1 - typ_1)

; Przykład 2 (:typing)
(:objects obiekt_1 obiekt_2 obiekt_3 - typ_1
          obiekt_4 - typ_2
          obiekt_5 - typ_3)
```

3.3. Stan początkowy

W sekcji *init* definiujemy stan początkowy za pomocą koniunkcji relacji i właściwości startowych.

```
; Przykład (:strips)
(:init (typ_1 obiekt_1)
       (typ_1 obiekt_2)
       (typ_2 obiekt_3)
       (Wlasciwosc_1 obiekt_1)
       (Relacja_1 obiekt_2 obiekt_3)
)
```

Stosując deklarację *typing* pomijamy przypisanie typów do obiektów, ponieważ następuje ono w elemencie *objects*. Sekcja *init* wygląda więc następująco:

```
; Przykład (:typing)
(:init (Wlasciwosc_1 obiekt_1)
       (Relacja_1 obiekt_2 obiekt_3))
```

3.4. Stan docelowy

Stan docelowy formułuje się identycznie jak warunek stosowalności akcji. Zdanie logiczne w elemencie *goal* określa zależności które chcemy uzyskać.

```
; Przykład 1 (:strips)
(:goal (Wlasciwosc_1(?obiekt_1)))

; Przykład 2 (:equality)
(:goal (= (?obiekt_1) (?obiekt_1)))

; Przykład 3 (:adl)
(:goal (forall (Wlasciwosc_1(?obiekt_X))
         (Relacja_1(?obiekt_1 ?obiekt_X)))

; Przykład 4 (:adl)
(:goal (exists (Wlasciwosc_1(?obiekt_X))
         (Relacja_1(?obiekt_1 ?obiekt_X)))
```

4. Algorytm Graphplan

4.1. Opis algorytmu

Graphplan jest algorytmem automatycznego planowania opracowanym w 1995 roku[3][4]. Algorytm przyjmuje jako wejście problem planowania wyrażony w systemie STRIPS i tworzy, jeśli jest to możliwe, sekwencję operacji umożliwiającą osiągnięcie stanu docelowego. Rozwiązanie uzyskiwane jest w procesie rozwijania grafu planowania, którego węzły przedstawiają możliwe stany, zaś krawędzie dopuszczalne połączenia.

Graf planowania

Graf planowania składa się z dwóch typów węzłów:

- Akcje (ang. actions)
- Propozycje (ang. propositions)

Poziomy parzyste grafu stanowią węzły propozycji, natomiast poziom zerowy propozycje prawdziwe dla stanu początkowego. Nieparzyste zaś składają się z akcji, których warunki stosowalności są obecne w poprzednim poziomie. W wyniku wykonywania akcji, w kolejnych poziomach otrzymujemy nowe propozycje.

W jednym kroku algorytmu tworzony jest jeden poziom węzłów akcji i propozycji.

Warunki wykluczania

Podczas rozbudowywania grafu planowania wyznaczane są wzajemnie wykluczające się relacje (ang. mutual exclusion relation „mutex") między węzłami tego samego poziomu. Warunki wykluczania definiuje się następująco:

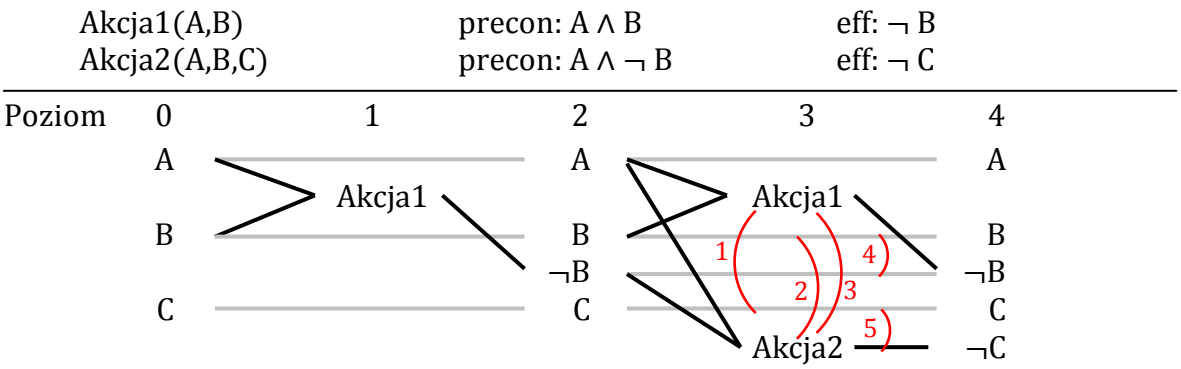
- Akcje wykluczają się, jeśli jedna akcja usuwa warunek stosowalności drugiej (ang. Intefrece/Inconsistent Effects)
- Akcje wykluczają się, jeśli jedna akcja posiada przeciwny warunek stosowalności niż druga (ang. Competing Needs)
- Propozycje wykluczają się, jeśli wszystkie metody ich osiągnięcia są wykluczeniami

Na przykładzie z rys. 1 Akcja1 i Akcja2 są wykluczeniem (1) ponieważ ich warunki stosowalności są sprzeczne ($B \wedge \neg B$).

Sprawdzanie osiągnięcia celu

Poszukiwanie rozwiązania metodą backtracking (ang. backtracking chain goal search) następuje za każdym razem gdy uzyskujemy nowy poziom węzłów propozycji. Cel zostaje osiągnięty jeśli jego warunki są obecne w aktualnym poziomie oraz gdy istnieje metoda ich osiągnięcia, która się nie wyklucza.

Przykład rozwijania grafu



Rys. 1. Ilustracja rozwijania grafu: kolor szary – akcje typu *no-operation (no-op)*, kolor czerwony – wzajemne wykluczanie.

4.2. Wykorzystywane narzędzie

Wykorzystywanym narzędziem opartym o wcześniej wspomnianą metodę poszukiwania rozwiązania jest Sensory Graphplan (SGP)[p-1][p-2].

Tab. 2. Informacje statystyczne dostępne w narzędziu SGP.

Dana statystyczna	Opis
time-expansion	Czas potrzebny na rozwinięcie grafu w każdym kroku algorytmu
time-bc	Czas potrzebny na sprawdzanie osiągnięcia celu po rozwinięciu grafu w każdym kroku algorytmu
time-mutex	Czas potrzebny na wyznaczenie wzajemnie wykluczających się relacji w każdym kroku algorytmu
graph-size	Liczba akcji i propozycji otrzymanych w każdym kroku algorytmu
mutex-count	Liczba wzajemnie wykluczających się relacji otrzymanych w każdym kroku algorytmu

5. Dziedzina „Gripper”

5.1. Opis dziedziny

„Gripper” jest prostym przykładem zadania planowania akcji robota transportowego. Przenoszone przedmioty reprezentowane są przez piłki, lokalizacje przez pokoje, natomiast obiektom załadunkowym odpowiadają chwytaki robota. Stan otoczenia opisany jest przez lokalizację piłek i robota oraz zawartość chwytaków. Zadaniem robota jest doprowadzić otoczenie ze stanu początkowego do stanu docelowego w jak najmniejszej liczbie kroków.

5.2. Domena – opis i implementacja

Predykaty

Predykaty domeny „Gripper” definiują dopuszczalne własności i zależności w otoczeniu:

- Położenie obiektów (piłki, robot)
- Stan obiektów (chwytaki)

Tab. 3. Predykaty domeny „Gripper”.

Predykat	Opis
room (r)	Obiekt *r* jest pokojem
ball (b)	Obiekt *b* jest piłką
gripper (g)	Obiekt *g* jest chwytakiem
at-robby (r)	Robot znajduje się w pomieszczeniu *r*
at (b, r)	Piłka *b* znajduje się w pokoju *r*
free (g)	Chwytak *g* jest wolny
carry (obj, g)	Chwytak *g* dźwiga obiekt *obj*

Akcje/operatory

Akcje domeny „Gripper” definiują dopuszczalne metody przekształcania otoczenia:

- Przemieszczanie robota
- Podnoszenie i opuszczanie przedmiotów przez chwytaki

Tab. 4. Akcje domeny „Gripper”.

Akcja	Opis
move (from, to)	Robot może poruszać się z pokoju *from* do pokoju *to*
	Warunek stosowalności: room (from) ∧ room (to) ∧ at-robby (from)
	Efekt: at-robby (to) ∧ ¬ at-robby (from)
pick (obj, room, gripper)	Robot może podnieść obiekt *obj* w pokoju *room* chwytakiem *gripper*
	Warunek stosowalności: ball (obj) ∧ room (room) ∧ gripper (gripper) ∧ at-robby (room) ∧ at (obj, room) ∧ free (gripper)
	Efekt: carry (obj, gripper) ∧ ¬ at (obj, room) ∧ ¬ free (gripper)
drop (obj, room, gripper)	Robot może odłożyć obiekt *obj* w pokoju *room* chwytakiem *gripper*
	Warunek stosowalności: ball (obj) ∧ room (room) ∧ gripper (gripper) ∧ at-robby (room) ∧ carry (obj, gripper)
	Efekt: at (obj, room) ∧ free (gripper) ∧ ¬ carry (obj, gripper)

Implementacja domeny „Gripper” w systemie STRIPS

```
(define (domain gripper-strips)
  (:requirements :strips)

  (:predicates (room ?r)
               (ball ?b)
               (gripper ?g)
               (at-robby ?r)
               (at ?b ?r)
               (free ?g)
               (carry ?o ?g))

  (:action move
      :parameters  (?from ?to)
      :precondition (and  (room ?from)
          (room ?to)
          (at-robby ?from))
      :effect (and  (at-robby ?to)
          (not (at-robby ?from))))

  (:action pick
      :parameters (?obj ?room ?gripper)
      :precondition  (and  (ball ?obj)
          (room ?room)
          (gripper ?gripper)
          (at ?obj ?room)
          (at-robby ?room)
          (free ?gripper))
      :effect (and (carry ?obj ?gripper)
          (not (at ?obj ?room))
          (not (free ?gripper))))

  (:action drop
      :parameters  (?obj  ?room ?gripper)
      :precondition  (and  (ball ?obj)
          (room ?room)
          (gripper ?gripper)
          (carry ?obj ?gripper)
          (at-robby ?room))
      :effect (and (at ?obj ?room)
          (free ?gripper)
          (not (carry ?obj ?gripper)))))
```

Implementacja domeny „Gripper” z wykorzystaniem deklaracji TYPING

```
(define (domain gripper-typed)
  (:requirements :typing)

  (:types room ball gripper)

  (:predicates (at-robby ?r - room)
               (at ?b - ball ?r - room)
               (free ?g - gripper)
               (carry ?o - ball ?g - gripper))

  (:action move
      :parameters  (?from ?to - room)
      :precondition (at-robby ?from)
      :effect (and  (at-robby ?to)
          (not (at-robby ?from))))

  (:action pick
      :parameters (?obj - ball ?room – room
          ?gripper - gripper)
      :precondition  (and  (at ?obj ?room)
          (at-robby ?room)
          (free ?gripper))
      :effect (and (carry ?obj ?gripper)
          (not (at ?obj ?room))
          (not (free ?gripper))))

  (:action drop
      :parameters (?obj - ball ?room – room
          ?gripper - gripper)
       :precondition  (and  (carry ?obj ?gripper)
          (at-robby ?room))
       :effect (and (at ?obj ?room)
          (free ?gripper)
          (not (carry ?obj ?gripper)))))
```

Deklaracja TYPING pozwala na redukcję liczby predykatów o te określające typy obiektów (room, ball, gripper). W rezultacie formuły warunków stosowalności akcji są krótsze niż w przypadku implementacji w systemie STRIPS.

5.3. Przykład problemu prostego – opis, implementacja, rozwiązanie i graf

Opis problemu

Problem: ‘gripper-1’

Obiekty:

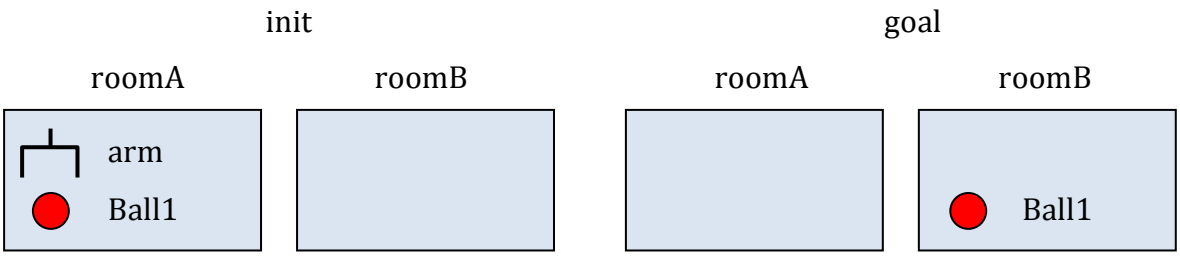
Ball1	- ball
roomA	- room
arm	- gripper

Stan początkowy:

at (Ball1, roomA)
at-robby (roomA)
free (arm)

Stan docelowy:

at (Ball1, roomB)



Rys. 2. Ilustracja problemu ‘gripper-1’.

Implementacja problemu ‘gripper-1’ w systemie STRIPS

```
(define (problem gripper-1s)
  (:domain gripper-strips)

  (:objects roomA roomB Ball1 arm)

  (:init
      (room roomA)
      (room roomB)
      (ball Ball1)
      (gripper arm)
      (at-robby roomA)
      (free arm)
      (at Ball1 roomA))

  (:goal (at Ball1 roomB))))
```

Implementacja problemu ‘gripper-1’ z wykorzystaniem deklaracji TYPING

```
(define (problem gripper-1t)
  (:domain gripper-typed)

  (:objects roomA - room Ball1 – ball arm - gripper)

  (:init
      (at-robby roomA)
      (free arm)
      (at Ball1 roomA))

  (:goal (at Ball1 roomB))))
```

Rozwiązanie w programie SGP

Rozwiązanie:

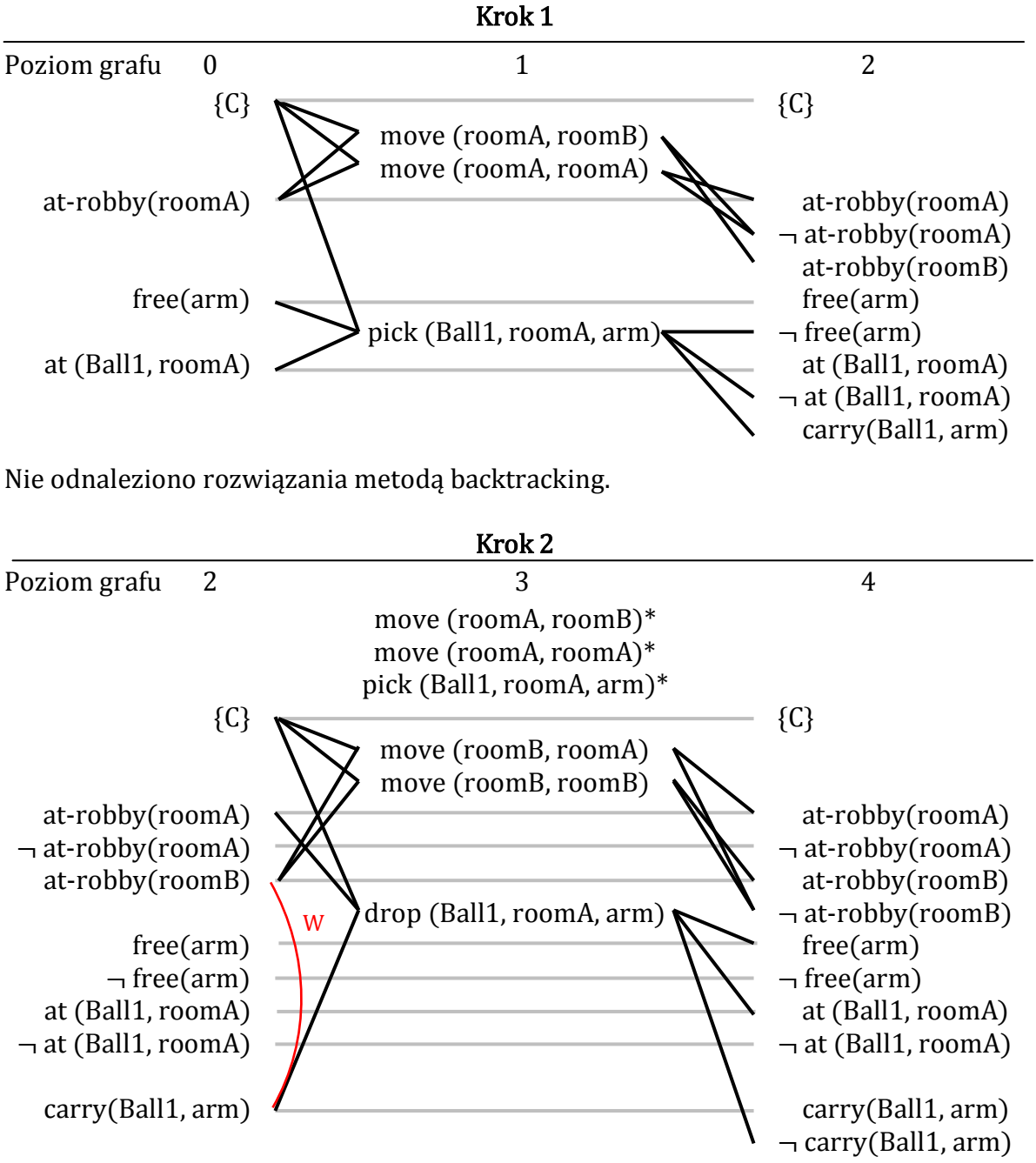
1 ° pick (Ball1, roomA, arm)
2 ° move (roomA, roomB)
3 ° drop (Ball1, roomB, arm)

Tab. 5. Przebieg rozwiązywania problemu ‘gripper-1’ w programie SGP.

Krok	time-expansion (s)	time-bc (s)	time-mutex (s)	graph-size (akcje, propozycje)	mutex-count (akcje, propozycje)
1	0,110	0,009	0,027	(13,15)	(18,36)
2	0,133	0,010	0,041	(21,15)	(138,24)
3	0,142	0,004	0,050	(22,16)	(142,32)
Σ	0,385	0,023	0,118		
Σ	0,408				

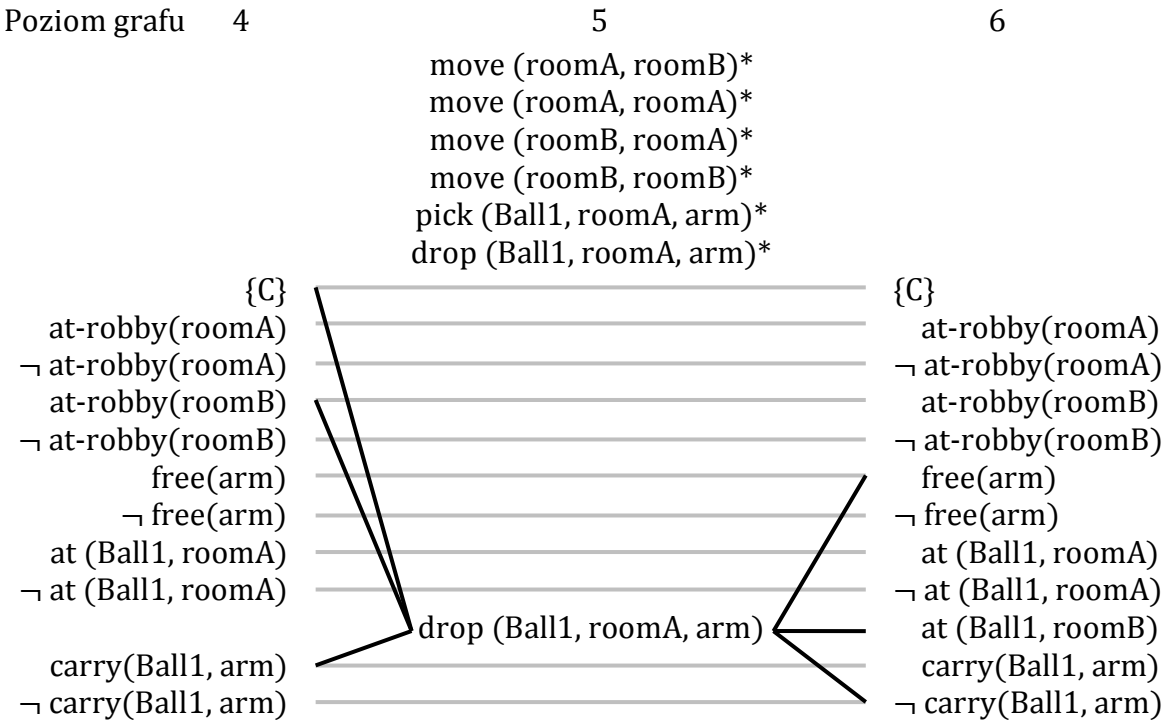
Graf rozwiązania

W celu przejrzystości graf rozwiązania przedstawiony na rys. 3 i 4 został uproszczony. Powtarzające się akcje na tym samym poziomie zostały pominięte, zaś stałe propozycje określające typy obiektów wyrażono przez zbiór C. Na grafie zaznaczono również jedynie najistotniejsze wzajemne wykluczenie i pominięto pozostałe.



Rys. 3. Uproszczona ilustracja rozwiązania problemu ‘gripper-1’ krok 1 i 2: kolor szary – akcje typu *no-operation (no-op)*, W – wykluczenie, dla którego warunek stosowalności akcji *drop (Ball1, roomB, arm)* jest sprzeczny, * – akcje opisano w poprzednich krokach algorytmu.

Krok 3



Odnaleziono rozwiązanie metodą backtracking:

3 °
at (Ball1, roomB) -> drop (Ball1, roomB, arm)
2 °
{C} -> no-op
carry (Ball1, arm) -> no-op
at-robby (roomB) -> move (roomA, roomB)
1 °
{C} -> no-op
at-robby (roomA) -> no-op
carry (Ball1, arm) -> pick (Ball1, roomA, arm)
0 °
{C}
at-robby (roomA)
free (arm)
at (Ball1, roomA)

Rozwiązanie:

1 ° pick (Ball1, roomA, arm)
2 ° move (roomA, roomB)
3 ° drop (Ball1, roomB, arm)

Rys. 4. Uproszczona ilustracja rozwiązania problemu ‘gripper-1’ krok 3: kolor szary – akcje typu *no-operation (no-op)*, * – akcje opisano w poprzednich krokach algorytmu.

5.4. Przykład problemu złożonego – opis i rozwiązanie

Opis problemu

Problem: ‘gripper-2’

Obiekty:

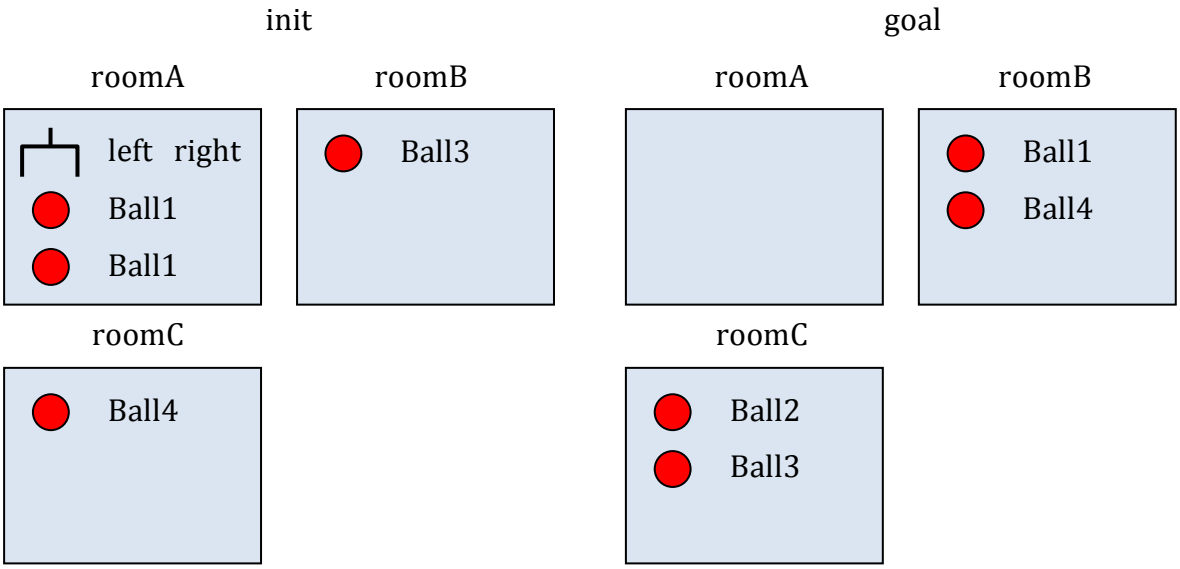
Ball1 Ball2 Ball3 Ball4	- ball
roomA roomB roomC	- room
left right	- gripper

Stan początkowy:

at (Ball1, roomA)
at (Ball2, roomA)
at (Ball3, roomB)
at (Ball4, roomC)
at-robby (roomA)
free (left)
free (right)

Stan docelowy:

at (Ball1, roomB)
at (Ball1, roomC)
at (Ball1, roomC)
at (Ball1, roomB)



Rys. 5. Ilustracja problemu ‘gripper-2’.

Rozwiązanie w programie SGP

Rozwiązanie:

1 ° (pick (Ball1, roomA, left) pick (Ball2, roomA, right))
2 ° move (roomA, roomC)
3 ° drop (Ball2, roomC, right)
4 ° pick (Ball4, roomC, right)
5 ° move (roomC, roomB)
6 ° (drop (Ball1, roomB, left) drop (Ball4, roomB, right))
7 ° pick (Ball3, roomB, left)
8 ° move (roomB, roomC)
9 ° drop (Ball3, roomC, left)

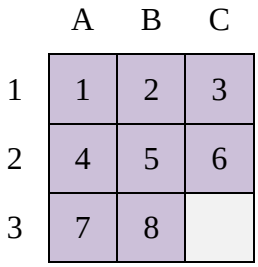
Tab. 6. Przebieg rozwiązywania problemu ‘gripper-2’ w programie SGP.

Krok	time-expansion (s)	time-bc (s)	time-mutex (s)	graph-size (akcje, propozycje)	mutex-count (akcje, propozycje)
1	0,590	0,002	0,170	(41,45)	(72,100)
2	0,993	0,002	0,298	(66,51)	(972,218)
3	1,337	0,014	0,446	(84,55)	(2214,206)
4	1,725	0,015	0,650	(104,59)	(3839,248)
5	1,991	0,030	0,746	(116,59)	(5024,182)
6	1,921	0,158	0,678	(116,59)	(4634,176)
7	1,917	0,676	0,682	(116,59)	(4604,176)
8	1,954	3,737	0,690	(116,59)	(4604,176)
9	1,933	2,874	0,689	(116,59)	(4604,176)
Σ	14,361	7,508	5,049		
Σ	21,869				

6. Dziedzina „Puzzle”

6.1. Opis dziedziny

„Puzzle” jest przykładem zadania polegającym na rozwiązywaniu układanki. Gra składa się z dwuwymiarowej planszy oraz z puzzli, których liczba jest o jeden mniejsza od liczby pól. Fragmenty układanki można przesuwać w poziomie lub pionie na przylegające puste pola. Celem jest dojście do rozwiązania z początkowego ułożenia fragmentów układanki w jak najmniejszej liczbie kroków.



Rys. 6. Ilustracja układanki z dziedziny „Puzzle”.

6.2. Domena – opis

Predykaty

Predykaty domeny „Puzzle” definiują dopuszczalne zależności w układance:

- Wzajemne położenia obiektów (pola, fragmenty układanki)
- Stan obiektów (pola)

Tab. 7. Predykaty domeny „Puzzle”.

Predykat	Opis
piece (p)	Obiekt *p* jest fragmentem układanki
square (s)	Obiekt *s* jest polem na planszy
adj (s1, s2)	Pola *s1* i *s2* stykają się ze sobą
at (p, s)	Fragment *p* znajduje się na polu *s*
empty (s)	Pole *s* jest puste

Akcje/operatory

Jedyną dopuszczalną metodą przekształcania układanki jest przemieszczanie jej fragmentów.

Tab. 8. Akcje domeny „Puzzle”.

Akcja	Opis
slide (p, from, to)	Fragment układanki *p* można przesuwać z pola *from* na pole *to*
	Warunek stosowalności: piece (p) ∧ square (from) ∧ square (to) ∧ empty (to) ∧ at (p, from) ∧ adj (from, to)
	Efekt: at (p, to) ∧ empty (from) ∧ ¬ empty (to)

6.3. Przykład problemu prostego – opis i rozwiązanie

Opis problemu

Problem: ‘puzzle-1’

Obiekty:

A1 A2 B1 B2 - square
P1 P2 P3 - piece

Stan początkowy:

at (P1, A1) at (P2, B1)
empty (A2) at (P3, B2)

Stan docelowy:

at (P1, A1) at (P2, B1)
at (P3, A2)

Rys. 7. Ilustracja problemu ‘puzzle-1’.

Rozwiązanie w programie SGP

Rozwiązanie:

1 ° slide (P1, A2, A1)
2 ° slide (P3, B2, A2)

Tab. 9. Przebieg rozwiązywania problemu ‘puzzle-1’ w programie SGP.

Krok	time-expansion (s)	time-bc (s)	time-mutex (s)	graph-size (akcje, propozycje)	mutex-count (akcje, propozycje)
1	0,773	0,025	0,190	(41,46)	(18,68)
2	1,009	0,027	0,243	(52,50)	(230,122)
Σ	1,782	0,052	0,433		
Σ	1,834				

6.4. Przykład problemu złożonego – opis i rozwiązanie

Opis problemu

Problem: ‘puzzle-1’

Obiekty:

A1 A2 A3 B1 B2 B3 C1 C2 C3 - square
P1 P2 P3 P4 P5 P6 P7 P8 - piece

Stan początkowy:

at (P2, A1) at (P4, B1) at (P3, C1)
at (P1, A2) at (P5, B2) at (P6, C2)
empty (A3) at (P7, B3) at (P8, C3)

Stan docelowy:

at (P1, A1) at (P2, B1) at (P3, C1)
at (P4, A2) at (P5, B2) at (P6, C2)
at (P7, A3) at (P8, B3)

Rys. 8. Ilustracja problemu ‘puzzle-2’.

Rozwiązanie w programie SGP

Rozwiązanie:

1 ° slide (P7, B3, A3)
2 ° slide (P5, B2, B3)
3 ° slide (P4, B1, B2)
4 ° slide (P2, A1, B1)
5 ° slide (P1, A2, A1)
6 ° slide (P4, B2, A2)
7 ° slide (P5, B3, B2)
8 ° slide (P8, C3, B3)

Tab. 10. Przebieg rozwiązywania problemu ‘puzzle-2’ w programie SGP.

Krok	time-expansion (s)	time-bc (s)	time-mutex (s)	graph-size (akcje, propozycje)	mutex-count (akcje, propozycje)
1	16,385	0,017	2,393	(181,186)	(18,68)
2	20,405	0,025	2,722	(194,196)	(300,308)
3	22,067	0,017	3,174	(214,206)	(1334,606)
4	24,089	0,027	3,744	(238,216)	(3346,908)
5	26,298	0,092	4,491	(266,225)	(6774,1296)
6	28,748	0,202	5,340	(295,231)	(11614,1368)
7	30,643	0,210	6,124	(321,235)	(16502,1254)
8	31,946	0,520	6,853	(345,241)	(21432,1278)
Σ	200,581	1,11	34,841		
Σ	201,691				

7. Podsumowanie

Niniejsza praca ma charakter dydaktyczny i prezentuje wybrane aspekty języka PDDL. Opisy elementów oraz przykładowe skrypty w niej zawarte mogą posłużyć jako materiały pomocnicze do nauki pisania zadań planowania w tym języku.

Wykorzystane narzędzie Sensory Graphplan umożliwia rozwiązywanie zadań napisanych w języku PDDL w opisanych podzbiorach.

Do rozwiązania opisanych zadań posłużył dwurdzeniowy procesor 2.26 GHz (Intel Core 2 Duo P8400). Wyniki w tabelach 5, 6, 9 i 10 pokazują jak zmienia się czas wyznaczania planu dla prostych i złożonych problemów.

Program w wykorzystanej wersji nakłada pewne ograniczenia uniemożliwiające rozwiązywanie bardziej złożonych przykładów. Plan generowany przez SGP składa się maksymalnie z 10 kroków, tak więc zadania wymagające większej liczby kroków nie są rozwiązywalne przy pomocy tego narzędzia. Ponadto nie jest ono kompatybilne z wszystkimi podzbiorami języka PDDL jak np. podzbiór aksjomatów (ang. axioms). W celu rozwiązania tego typu zadań zaleca się skorzystanie z bardziej zaawansowanego narzędzia planowania.

8. Literatura

[1] Stuart J. Russell, Peter Norvig: Artificial Intelligence A Modern Approach. New Jersey: Prentice-Hall, Inc. 1995. ISBN 0-13-103805-2.

[2] Drew McDermott: PDDL – The Planning Domain Definition Language. Yale University 1998.

[3] David E. Smith, Daniel S. Weld: Conformant Graphplan. Proc. 15th Nat. Conf. Artificial Intelligence 1998, Madison, Wisconsin, USA, s. 889–896.

[4] Daniel S. Weld, Corin R. Anderson, David E. Smith: Extending Graphplan to handle Uncertainty & Sensing Actions. Proc. 15th Nat. Conf. Artificial Intelligence 1998, Madison, Wisconsin, USA, s. 897–904.

[5] Writing Planning Domains and Problems in PDDL [online]. [dostęp 15 stycznia 2010]. Dostępny w Internecie: < http://vir.liu.se/~TDDC17/info/labs/planning/2004 /writing.html>

Wykorzystane narzędzia

[p-1] Sensory Graph Plan (SGP) ver. 1.0h Lisp implementation © Mark Peot, Dave Smith, Dave Smith, Dan Weld, Corin Anders

[p-2] Allegro Common Lisp (ACL) ver. 8.2 © Franz Inc.

Dodatek A

Dodatek A zawiera opis obsługi narzędzia Sensory Graphplan (SGP) w środowisku Allegro Common Lisp.

Uruchamianie narzędzia SGP

Komenda	Przykład
:cd SCIEZKA_FOLDERU_SGP :cl loader.lisp (in-package :cl-user) (load-gp)	:cd C:\SGP

Wczytanie pliku *.pddl

Plik *.pddl powinien znajdować się w folderze .\Domains

Komenda	Przykład
(load-domains "NAZWA_PLIKU")	(load-domains "gripper.pddl")

Generowanie planu

Komenda	Przykład
(plan 'NAZWA_PLANU)	(plan 'gripper-1)

Włączenie/wyłączenie wyświetlania dodatkowych informacji planu

Komenda	Przykład
(stat-gp DANA_STATYSTYCZNA)	(stat-gp time-expansion) (stat-gp time-bc) (stat-gp time-mutex) (stat-gp graph-size) (stat-gp mutex-count)
(trace-gp DANA_ŚLEDZONA)	(trace-gp actions) (trace-gp mutex) (trace-gp bc)