Uniwersytet im. Adama Mickiewicza w Poznaniu
Wydział Matematyki i Informatyki

Aleksandra Ratajczak
Nr albumu: **340951**

Indukcyjne programowanie w logice
Inductive Logic Programming

Praca magisterska na kierunku:
Informatyka

Specjalność:
Systemy inteligentne

Promotor:
prof. UAM dr hab. Krzysztof Jassem

Poznań, 2017

Poznań, dnia .....

(data)

Oświadczenie

Ja, niżej podpisana **Aleksandra Ratajczak**, studentka Wydziału Matematyki i Informatyki Uniwersytetu im. Adama Mickiewicza w Poznaniu, oświadczam, że przedkładaną pracę dyplomową pt. ***"Indukcyjne programowanie w logice"*** napisałam samodzielnie. Oznacza to, że przy pisaniu pracy, poza niezbędnymi konsultacjami, nie korzystałam z pomocy innych osób, a w szczególności nie zlecałam opracowania rozprawy lub jej części innym osobom, ani nie odpisywałam tej rozprawy lub jej części od innych osób.

Oświadczam również, że egzemplarz pracy dyplomowej w wersji drukowanej jest całkowicie zgodny z egzemplarzem pracy dyplomowej w wersji elektronicznej.

Jednocześnie przyjmuję do wiadomości, że gdyby powyższe oświadczenie okazało się nieprawdziwe, decyzja o wydaniu mi dyplomu zostanie cofnięta.

Jednocześnie przyjmuję do wiadomości, że przypisanie sobie, w pracy dyplomowej, autorstwa istotnego fragmentu lub innych elementów cudzego utworu lub ustalenia naukowego stanowi podstawę stwierdzenia nieważności postępowania w sprawie nadania tytułu zawodowego.

[**TAK**]* - wyrażam zgodę na udostępnianie mojej pracy w czytelni Archiwum UAM

[**TAK**]* - wyrażam zgodę na udostępnianie mojej pracy w zakresie koniecznym do ochronymojego prawa do autorstwa lub praw osób trzecich

* Należy wpisać TAK w przypadku wyrażenia zgody na udostępnianie pracy w czytelni Archiwum UAM, NIE w przypadku braku zgody. Niewypełnienie pola oznacza brak zgody na udostępnianie pracy.

.....

(czytelny podpis studenta)

Streszczenie

Niniejsza praca opisuje zagadnienie indukcyjnego programowania w logice (IPL) i dostarcza bazową wiedzę potrzebną do jego zrozumienia. Motywacją, jaką kieruje się autorka, jest rozszerzenie eksperymentu przedstawionego w artykule *Inductive Logic Programming With Large-Scale Unstructured Data.* Interesującym aspektem rozwiązania modelowego jest znalezienie optymalnej rozgrywki dla gry końcowej z udziałem białego króla i białej wieży przeciwko czarnemu królowi — ang. *King and Rook against King endgame (KRK).* Do dedukcji reguł użyte zostało narzędzie GOLEM przygotowane przez profesora Stephena H. Muggletona. Dane dostarczone z eksperymentem modelowym zostały użyte także w rozwiązaniu autorskim. Praca krok po kroku objaśnia proces odtwarzania eksperymentu i jego rozszerzania.

Spis treści

1 Wstęp

1.1 Motywacja

Zagadnieniem rozpatrywanym w niniejszej pracy jest indukcyjne programowanie w logice z wykorzystaniem skończonych, obszernych danych. Celem pracy jest wykorzystanie indukcyjnego programowania w logice (IPL) dla rozgrywek szachowych poprzez odtworzenie i rozszerzenie problemu zamodelowanego w pracy *Inductive Logic Programming With Large-Scale Unstructured Data* [BS94].

1.2 Hipoteza badawcza

Rozwiązanie modelowe przedstawione w [BS94] ma zostać rozszerzone o predykat znajdujący optymalną rozgrywkę dla gry końcowej KRK.

1.3 Układ pracy

Rozdział 1. wyjaśnia motywacje i cele oraz przedstawia ogólny układ pracy.

W rozdziale 2. omówione zostało samo zagadnienie indukcyjnego programowania w logice wraz z objaśnieniem podstaw matematycznych (rozdziały 2.1. i 2.2.) oraz używanej w przykładach i definicjach notacji języka Prolog (rozdział 2.3.).

Rozdział 3. przedstawia eksperyment wzorcowy, na którym opiera się eksperyment autorski.

W rozdziale 4. opisany został przeprowadzony eksperyment autorski wraz ze szczegółami dotyczącymi modyfikacji wzorcowego rozwiązania.

Rozdział 5. zawiera wnioski dotyczące znalezionego rozwiązania oraz otrzymanych wyników.

2 Indukcyjne programowanie w logice

2.1 Klasyczny rachunek zdań

Wprowadzamy podstawowe pojęcia klasycznego rachunku zdań (KRZ).

2.1.1 Spójniki logiczne

Definicja 2.1. *Zdaniem w sensie logicznym* nazywamy wyrażenie, które przyjmuje jedną z wartości logicznych: *prawdę* lub *fałsz*. Prawdę oznaczamy symbolem 1, a fałsz symbolem 0.

Definicja 2.2. *Zmienna zdaniowa* reprezentuje dowolne zdanie. Zmienne zdaniowe zwyczajowo oznaczamy małymi literami, także z indeksami.

Przykład.
Oznaczeniami zmiennych zdaniowych są na przykład: p, q, r_1, r_2, r_3.

Definicja 2.3. *Spójniki logiczne* (inaczej: *funktory KRZ*) służą do konstrukcji zdań logicznie złożonych. Wyróżniamy pięć podstawowych spójników logicznych:

($\neg$) negację,

($\wedge$) koniunkcję,

($\vee$) alternatywę,

($\Rightarrow$) implikację,

($\Leftrightarrow$) równoważność.

Spójniki logiczne przyjmują jednoznacznie określoną ilość argumentów. Spójnik negacji jest jednoargumentowy. Spójniki koniunkcji, alternatywy, implikacji oraz równoważności są dwuargumentowe.

Spójniki logiczne są **ekstensjonalne**, tj. wartość logiczna zdania złożonego za pomocą danego spójnika jest jednoznacznie wyznaczona przez ten spójnik i wartości logiczne argumentów, jakie przyjmuje.

Definicja 2.4. *Spójnik **negacji*** jest logicznym odwzorowaniem stwierdzenia „nieprawda, że" użytym w kontekście zdania „nieprawda, że p". Negację oznaczamy symbolem $\neg$. Zdanie postaci $\neg p$ nazywamy *negacją zdania p*.

Negacja zdania prawdziwego jest zdaniem fałszywym. Negacja zdania fałszywego jest zdaniem prawdziwym.

Definicja 2.5. *Spójnik **koniunkcji*** jest logicznym odwzorowaniem stwierdzenia „i" użytym w kontekście zdania „p i q". Koniunkcję oznaczamy symbolem $\wedge$. Zdanie postaci $p \wedge q$ nazywamy *koniunkcją zdań p i q*.

Koniunkcja dwóch zdań jest prawdziwa wtedy i tylko wtedy, gdy oba argumenty są prawdziwe.

Definicja 2.6. *Spójnik* ***alternatywy*** jest logicznym odwzorowaniem stwierdzenia „lub" użytym w kontekście zdania „p lub q". Alternatywę oznaczamy symbolem $\vee$. Zdanie postaci $p \vee q$ nazywamy *alternatywą zdań p i q*.

Alternatywa dwóch zdań jest prawdziwa wtedy i tylko wtedy, gdy przynajmniej jeden argument jest prawdziwy.

Definicja 2.7. *Spójnik* ***implikacji*** jest logicznym odwzorowaniem stwierdzenia „jeżeli..., to..." użytym w kontekście zdania „jeżeli p, to q". Implikację oznaczamy symbolem $\Rightarrow$. Zdanie postaci $p \Rightarrow q$ nazywamy *implikacją o* ***poprzedniku*** *p i* ***następniku*** *q.*

Implikacja dwóch zdań (zdanie warunkowe) jest fałszywa wtedy i tylko wtedy, gdy poprzednik jest prawdziwy, a następnik jest fałszywy. Inaczej: z prawdy nie może logicznie wynikać fałsz.

Definicja 2.8. *Spójnik* ***równoważności*** jest logicznym odwzorowaniem stwierdzenia „wtedy i tylko wtedy, gdy" użytym w kontekście zdania „p wtedy i tylko wtedy, gdy q". Równoważność oznaczamy symbolem $\Leftrightarrow$. Zdanie postaci $p \Leftrightarrow q$ nazywamy *równoważnością zdań p i q*.

Równoważność dwóch zdań jest prawdziwa wtedy i tylko wtedy, gdy oba argumenty mają tę samą wartość logiczną.

2.1.2 Tablice prawdziwościowe spójników KRZ

Poniższe tablice 2.1. i 2.2. przedstawiają wartości logiczne zdań złożonych skonstruowanych przy pomocy spójników logicznych KRZ. Zmienne zdaniowe przyjmowane jako argumenty funktorów oznaczamy jako p i q.

p	$\neg p$
0	1
1	0

Tablica 2.1: Tablica prawdziwościowa jednoargumentowego spójnika **negacji**.

p	q	$p \wedge q$	$p \vee q$	$p \Rightarrow q$	$p \Leftrightarrow q$
0	0	0	0	1	1
0	1	0	1	1	0
1	0	0	1	0	0
1	1	1	1	1	1

Tablica 2.2: Tablica prawdziwościowa dwuargumentowych spójników: **koniunkcji**, **alternatywy**, **implikacji** oraz **równoważności**.

2.1.3 Formuły KRZ

Definicja 2.9. *Formuła KRZ* to wyrażenie poprawnie zbudowane ze zmiennych zdaniowych za pomocą spójników logicznych. W formułach zawierających wiele spójników logicznych stosujemy nawiasy, żeby jednoznacznie określić priorytet każdego spójnika.

Przykład.
Wyrażenie $p \wedge q \vee r$ nie jest jednoznaczną formułą. Wprowadzając nawiasy otrzymujemy dwie istotnie różne formuły: $(p \wedge q) \vee r$ oraz $p \wedge (q \vee r)$.

Definicja 2.10. *Siłę wiązania spójników (**priorytety**)* określamy następująco, kolejno od najsilniejszych:

$\neg$ negacja,

$\wedge$**,** $\vee$ koniunkcja i alternatywa (równosilne),

$\Rightarrow$**,** $\Leftrightarrow$ implikacja i równoważność (równosilne).

Tak zdefiniowane priorytety pozwalają pominąć niektóre nawiasy w formule.

Definicja 2.11. Niech V będzie pewnym zbiorem zmiennych zdaniowych. *Wartościowaniem* w dla zbioru V nazywamy dowolną funkcję $w : V \mapsto \{0, 1\}$.

Każde wartościowanie w zbioru V jednoznacznie określa wartość logiczną dowolnej formuły KRZ, której zmienne należą do V.

Literami φ, ψ, χ, ... oznaczamy dowolne formuły KRZ. Wartość logiczną formuły φ dla wartościowania w oznaczamy przez $w(\varphi)$. Tę wartość można wyznaczyć przez następujące warunki rekurencyjne:

$$w(\neg\varphi) = \neg' w(\varphi),$$
$$w(\varphi \wedge \psi) = w(\varphi) \wedge' w(\psi),$$
$$w(\varphi \vee \psi) = w(\varphi) \vee' w(\psi),$$
$$w(\varphi \Rightarrow \psi) = w(\varphi) \Rightarrow' w(\psi),$$
$$w(\varphi \Leftrightarrow \psi) = w(\varphi) \Leftrightarrow' w(\psi).$$

Spójnik logiczny oznaczony symbolem $'$ rozumiemy jako zależny od wartościowania w dla danego wyrażenia.

Przykład.
Dana jest formuła logiczna $\varphi = p \vee q \Rightarrow p \wedge q$ oraz wartościowania $w(p) = 1$, $w(q) = 0$. Obliczamy:

$$\begin{aligned} w(\varphi) &= w(p \vee q) \Rightarrow' w(p \wedge q) \\ &= (w(p) \vee' w(q)) \Rightarrow' (w(p) \wedge' w(q)) \\ &= (1 \vee' 0) \Rightarrow' (1 \wedge' 0) \\ &= 1 \Rightarrow' 0 = 0 \end{aligned}$$

Jeżeli znane są wartości logiczne wszystkich składowych zmiennych zdaniowych, to w powyższy sposób możemy wyznaczyć wartość logiczną dowolnego zdania logicznie złożonego.

2.2 Klasyczny rachunek predykatów

Zagadnienie klasycznego rachunku predykatów (KRP) rozważamy w celu przedstawienia podstawowych pojęć w zakresie, w jakim jest ono wykorzystywane w języku Prolog. Klasyczny rachunek predykatów jest nazywany również: *rachunkiem predykatów pierwszego rzędu*, *rachunkiem kwantyfikatorów* lub *logiką pierwszego rzędu*, ang. *first-order logic (FOL), first-order predicate calculus*.

2.2.1 Klasyczny rachunek predykatów a klasyczny rachunek zdań

Klasyczny rachunek predykatów (KRP) rozszerza klasyczny rachunek zdań (KRZ) o relacje i kwantyfikatory. W KRP korzystamy ze spójników logicznych zdefiniowanych przez KRZ.

2.2.2 Symbole KRP

Definicja 2.12. *Zmienne indywiduowe* reprezentują elementy pewnej dziedziny obiektów, będącej niepustym zbiorem.

Przykład.
Oznaczeniami zmiennych indywiduowych są na przykład: x, y, z_1, z_2.

Definicja 2.13. *Stałe indywiduowe* oznaczają wyróżnione elementy dziedziny.

Przykład.
Oznaczeniami stałych indywiduowych są na przykład: a, b, c_1, c_2.

Definicja 2.14. *Stałe logiczne* są to spójniki logiczne KRZ oraz kwantyfikatory:

$\forall$ kwantyfikator *ogólny* (*generalny*, *uniwersalny*, *duży*),

$\exists$ kwantyfikator *szczegółowy* (*egzystencjalny*, *istnienia*, *mały*).

Kwantyfikatory zawsze występują razem ze zmienną. Czytamy je jako:

$\forall x$ „dla każdego x...”

$\exists x$ „istnieje x takie, że...”

Przykład.
$\forall x \exists y (x < y)$ czytamy: „dla każdego x istnieje y takie, że x jest mniejsze od y”.

Definicja 2.15. *Symbole relacyjne* (predykatowe) oznaczają relacje o jednoznacznie określonej liczbie argumentów (inaczej: *arności* lub *argumentowości*). Arność symboli relacyjnych oznaczamy poprzez indeks górny, np. P^1, Q^2, R^4. Argumentami symboli relacyjnych mogą być zmienne i stałe indywiduowe (również *termy*, zob. def. 2.17.). Symbol relacyjny razem z argumentami tworzy *formułę atomową* (atom — patrz def. 2.18).

Przykład.
Dane są symbole relacyjne P^1, Q^2, stała a oraz zmienne indywiduowe x, y. Można utworzyć następujące formuły atomowe:

$P(a)$ czytamy: „P od a" lub: „a ma własność P"

$Q(x, y)$ czytamy: „Q od x, y" lub: „x jest w relacji Q z y"

Jednoargumentowe symbole relacyjne reprezentują *własności* elementów dziedziny, dwuargumentowe - *stosunki dwuczłonowe* między elementami dziedziny (np. symbole matematyczne $=$, $<$, $\leq$), natomiast wieloargumentowe - *stosunki wieloczłonowe*.

Zwyczajowo symbole relacyjne oznaczamy wielką literą.

Definicja 2.16. *Symbole funkcyjne* reprezentują *operacje* (*działania*) określone na dziedzinie. Każdy symbol funkcyjny przyjmuje jednoznacznie określoną liczbę argumentów. Arność symboli funkcyjnych określamy tak samo jak arność symboli relacyjnych - za pomocą indeksu górnego, np. f^1, g^2, h^3.

Przykład.
f, g, h, $+$, $-$, $\times$.

Zwyczajowo symbole funkcyjne oznaczamy małą literą.

2.2.3 Podstawowe wyrażenia KRP

Definicja 2.17. *Term* to wyrażenie poprawnie zbudowane ze zmiennych i stałych indywiduowych za pomocą symboli funkcyjnych. Spośród termów wyróżniamy:

termy proste czyli zmienne i stałe indywiduowe,

termy złożone postaci $f(t_1, \cdots, t_n)$, gdzie f^n jest n-argumentowym symbolem funkcyjnym, a $t_1, \cdots, t_n$ są termami.

Przykład.
Termami są przykładowe wyrażenia: a, b, x, y, $f(a)$, $g(x, y)$, $h(a, f(x, y))$.

Definicja 2.18. *Formuła atomowa* (atom) to wyrażenie $P(t_1, \cdots, t_n)$ takie, że P^n jest n-argumentowym symbolem relacyjnym, a $t_1, \cdots, t_n$ są termami.

Przykład.
Formułami atomowymi są przykładowe wyrażenia: $P(a)$, $Q(x)$, $R(x, y)$, $S(f(x), a)$.

Definicja 2.19. *Literałami* nazywamy atomy i ich negacje. Literały dzielimy na:

pozytywne czyli formuły atomowe,

negatywne czyli negacje atomów.

Przykład.
Literałami są przykładowe wyrażenia: $P(a), Q(x,y), \neg P(a), \neg Q(x,y)$.

Definicja 2.20. *Formuła* to wyrażenie poprawnie zbudowane z formuł atomowych za pomocą spójników KRZ oraz kwantyfikatorów. Przyjmujemy siłę wiązania spójników logicznych jak w KRZ oraz nadajemy kwantyfikacjom $\forall x$, $\exists x$ tę samą siłę, co negacji. Formułę nazywamy *złożoną* jeśli jest jednej z postaci:

$$(\neg\varphi), (\varphi \wedge \psi), (\varphi \vee \psi), (\varphi \Rightarrow \psi), (\varphi \Leftrightarrow \psi), (\forall_x \varphi), (\exists_x \varphi)$$

gdzie φ, ψ są formułami.

Przykład.
Poprawnie zbudowaną formułą jest na przykład:
$\forall x P(x) \wedge \forall x Q(x) \Leftrightarrow \forall x (P(x) \wedge Q(x))$

Formuły nazywamy **wyrażeniami zdaniowymi**, a termy **wyrażeniami nazwowymi**. Termy i atomy nazywamy **wyrażeniami prostymi**.

2.2.4 Pojęcie klauzuli

Definicja 2.21. *Klauzula* to alternatywa skończenie wielu literałów, która ma postać:

$$\neg A_1 \vee \cdots \vee \neg A_n \vee B_1 \vee \cdots \vee B_m$$

gdzie $n, m \geq 0$, a A_i, B_j $(0 \geq i \geq n, 0 \geq j \geq m)$ są atomami. Powyższa klauzula jest logicznie równoważna implikacji:

$$A_1 \wedge \cdots \wedge A_n \Rightarrow B_1 \vee \cdots \vee B_m$$

Definicja 2.22. *Klauzula Horna* to klauzula zawierająca dokładnie jeden literał pozytywny:

$$\neg A_1 \vee \cdots \vee \neg A_n \vee B$$

albo

$$A_1 \wedge \cdots \wedge A_n \Rightarrow B$$

gdzie $n \geq 1$. B nazywamy *głową* (nagłówkiem, ang. *head*), a listę $A_1, \cdots, A_n$ nazywamy *treścią reguły* (ciałem, ang. *body*).

2.2.5 Unifikacja

Definicja 2.23. *Podstawieniem* σ nazywamy skończoną listę

$$\sigma = [x_1/t_1, \cdots, x_n/t_n]$$

taką, że $x_1, \cdots, x_n$ są różnymi zmiennymi indywiduowymi, a $t_1, \cdots, t_n$ są termami.

Definicja 2.24. Niech S będzie niepustym zbiorem wyrażeń prostych. Podstawienie σ takie, że $E\sigma = E'\sigma$ dla wszystkich $E, E' \in S$, nazywamy **podstawieniem uzgadniającym** (*unifikatorem*, ang. *unifier*) zbioru S. Podstawienie σ nazywamy *najogólniejszym podstawieniem uzgadniającym* (**najogólniejszym unifikatorem**, ang. *most general unifier*) zbioru S, jeżeli σ jest podstawieniem uzgadniającym zbioru S oraz dla każdego podstawienia uzgadniającego η zbioru S istnieje podstawienie γ takie, że $\eta = \sigma\gamma$.

Unifikator oznaczamy: $S\sigma = \{E\sigma : E \in S\}$ i czytamy „σ jest unifikatorem zbioru S" lub „σ unifikuje (uzgadnia) zbiór S".

Definicja 2.25. Zbiór S nazywamy *uzgadnialnym* (**unifikowalnym**), jeżeli istnieje unifikator zbioru S.

Przykład.
Dany jest zbiór $S = \{f(x,a), f(g(y,a),z)\}$ oraz podstawienia uzgadniające:

1) $\sigma = [x/g(y,a), z/a]$**,** mamy: $S\sigma = \{f(g(y,a),a)\}$.

2) $\sigma' = [x/g(a,a), y/a, z/a]$**,** mamy: $S\sigma = \{f(g(a,a),a)\}$.

σ' nie jest najogólniejszym unifikatorem zbioru S, ponieważ $\sigma' = \sigma[y/a]$. Zatem σ jest najogólniejszym unifikatorem zbioru S.

Definicja 2.26. *Wariantami* nazywamy wyrażenia proste E_1, E_2, jeżeli istnieją podstawienia σ_1, σ_2 takie, że $E_2 = E_1\sigma_1$ i $E_1 = E_2\sigma_2$.

Fakt 2.26.1 *Jeżeli σ_1, σ_2 są najogólniejszymi unifikatorami zbioru S, to $S\sigma_1$, $S\sigma_2$ są wariantami.*

Definicja 2.27. *Przemianowaniem zmiennych* w wyrażeniu E nazywamy podstawienie σ, jeżeli σ jest bijekcją zbioru $V(E)$ na zbiór $V(E\sigma)$.

Fakt 2.27.1 *Jeżeli wyrażenia proste E_1, E_2 są wariantami, to istnieją podstawienia η_1, η_2 takie, że $E_2 = E_1\eta_1$, $E_1 = E_2\eta_2$ oraz η_i jest przemianowaniem zmiennych w E_i dla $i = 1, 2$.*

Definicja 2.28. Niech $S = \{E_1, \cdots, E_n\}$, $n \geq 2$. Określamy *zbiór niezgodności* zbioru S, oznaczany przez D_S. Traktując wyrażenia E_i jako łańcuch symboli, znajdujemy najmniejszą pozycję l taką, że istnieją dwa wyrażenia E_i, E_j mające różne symbole na pozycji l. W każdym wyrażeniu z S na pozycji l musi występować symbol oznaczający: zmienną, stałą, symbol funkcyjny lub symbol relacyjny. Zbiór D_S składa się ze wszystkich podwyrażeń sensownych (termów lub atomów) e_i wyrażeń E_i, które zaczynają się na pozycji l.

Przykład.
Dany jest zbiór $S = \{f(x,a), f(g(y,a),z)\}$.

Uwzględniając syntaktyczną budowę termów, możemy wyobrazić sobie elementy zbioru S jako drzewa:

(a) $f(x, a)$ (b) $f(g(y, a), z)$

Rysunek 1: Drzewiasta reprezentacja elementów zbioru S

Wyrażenia z tego zbioru mają różne symbole na pierwszej pozycji symbolu funkcyjnego f. Zatem: $D_S = \{x, g(y, a)\}$.

Zauważmy, że x jest zmienną, a $g(y, a)$ jest termem nie zawierającym zmiennej x.

Fakt 2.28.1 *Jeżeli σ unifikuje zbiór S (przynajmniej dwuelementowy), to σ unifikuje zbiór D_S.*

Fakt 2.28.2 *Jeżeli zbiór D_S jest unifikowalny, to zawiera dwa wyrażenia, z których jedno jest zmienną, a drugie jest termem nie zawierającym tej zmiennej.*

Algorytm 2.1 *Algorytm unifikacji* (J. Robinson, 1965)

Dane wejściowe: $S = \{E_1, \cdots, E_n\}$, $n \geq 1$, E_i - wyrażenia proste.

1: $\sigma_0 = \epsilon$
2: **for** $k = 0, 1, 2, \cdots$ **do**
3: wyznacz zbiór $S\sigma_k$
4: **if** $|S\sigma_k| = 1$ **then**
5: **return** σ_k
6: **end if**
7: wyznacz $D_{S\sigma_k}$ ▷ dla $|S\sigma_k| > 1$
8: **if** nie istnieją $x, t \in D_{S\sigma_k}$ takie, że $x \notin V(t)$ **then**
9: **return** false
10: **else**
11: wybierz $x, t \in D_{S\sigma_k}$ takie, że $x \notin V(t)$
12: $\sigma_{k+1} \leftarrow \sigma_k[x/t]$
13: $k \leftarrow k + 1$
14: **end if**
15: **end for**

2.3 Język Prolog

Prolog jest komputerowym językiem programowania, który służy do opisywania oraz rozwiązywania problemów związanych z **obiektami** i formalnymi **relacjami** pomiędzy nimi.

Obiekty w języku Prolog różnią się znacznie od obiektów definiowanych przez paradygmat obiektowy. Nie są one strukturami danych z polami i metodami, ale *bytami*, które można opisywać *termami*.

Programowanie w Prolog można opisać jako **deklaratywne** i **opisowe**. Składa się ono z następujących części:

— deklarowania faktów dotyczących obiektów i związków między nimi,

— definiowania reguł dotyczących obiektów i związków między nimi,

— formułowania zapytań o obiekty i związki między nimi.

2.3.1 Podstawowe pojęcia

Definicja 2.29. *Predykatem* nazywamy symbol relacyjny o jednoznacznie określonej liczbie argumentów (*arności*).

Przykład. (Prolog)
```
kobieta/1.   % jednoargumentowy predykat kobieta
rodzic/2.    % dwuargumentowy predykat rodzic
```

Definicja 2.30. *Fakt* jest to zawsze prawdziwy związek pomiędzy obiektami. Każdy fakt zapisujemy w osobnym wierszu zakończonym kropką.

Przykład. (Prolog)
```
kobieta(iwona).          % Iwona jest kobietą
rodzic(aniela, iwona).   % Aniela jest rodzicem Iwony
rodzic(iwona, ola).      % Iwona jest rodzicem Oli
```

Definicja 2.31. *Reguła* jest to klauzula Horna z głową (ang. *head*) definiująca pewną relację (predykat). Regułę zapisujemy rozdzielając jej głowę od ciała symbolem `:-` i kończąc kropką. Składowe treści reguły rozdzielamy przecinkami. Przecinki czytamy jako „i" oraz utożsamiamy z koniunkcją.

Przykład. (Prolog)
Zapis:

```
matka(X, Y) :- kobieta(X), rodzic(X, Y).
babka(X, Y) :- kobieta(X), rodzic(X, Z), rodzic(Z,Y).
```

odpowiada następującym klauzulom Horna:

$$K(x) \wedge R(x,y) \Rightarrow M(x,y)$$

$$K(x) \wedge R(x,z) \wedge R(z,y) \Rightarrow B(x,y)$$

gdzie symbol relacyjny K^1 oznacza relację bycia kobietą, R^2 - bycia rodzicem, M^2 - bycia matką, a B^2 bycia babką.

Definicja 2.32. *Zapytaniem* języka Prolog nazwiemy pytanie dotyczące faktów i reguł rozpoczynające się od znaku zachęty `?-` i zakończone kropką. Formułując zapytania możemy posługiwać się zmiennymi. Odpowiedzią na zapytanie jest wartość logiczna, obiekt lub zestaw obiektów. W zależności od zadanego zapytania oraz dostarczonej bazy wiedzy (faktów i reguł) odpowiedzi na zapytanie może być wiele. Są one w takim przypadku oddzielone średnikiem.

Przykład. (Prolog)
Załóżmy zdefiniowane fakty jak w przykładzie w definicji 2.30. Zapytaniami są na przykład:

```
?- rodzic(aniela, iwona).
Yes

?- rodzic(ola, iwona).
No

?- rodzic(X, ola).
X = iwona

?- rodzic(X, Y).
X = aniela,
Y = iwona ;
X = iwona,
Y = ola.
```

Interpreter języka Prolog (np. SWI-Prolog) najczęściej od razu po uruchomieniu wyświetla znak zachęty i czeka na podanie zapytania.

2.3.2 Składnia języka Prolog

Definicja 2.33. *Termem* w języku Prolog określamy zmienną, atom (stałą lub liczbę) oraz obiekt złożony (strukturę).

Stała może być atomem lub liczbą. *Atom* jest to:

— obiekt, który definiujemy poprzez określenie na nim faktów oraz relacji jego dotyczących,

— łańcuch znaków.

Według konwencji stałe oznaczające symbole zapisujemy rozpoczynając nazwę małą literą.

Przykład. (Prolog)
Przykładowymi stałymi są:

```
iwona      % symbol
'Iwona'    % łańcuch znaków
-1         % liczba
3.14       % liczba
6e-3       % liczba
```

Zmienne zapisujemy rozpoczynając ich nazwę wielką literą lub znakiem podkreślenia.

Przykład. (Prolog)
Przykładowymi zmiennymi są:

```
X
X_1
Zmienna
_zmienna
_1
_
```

Ostatni z przykładów (pojedynczy znak podkreślenia _) oznacza **zmienną anonimową**. Wykorzystujemy ją, gdy nie interesuje nas wynik w postaci konkretnego obiektu, a tylko prawdziwość zapytania. Eliminuje to niewykorzystywane nigdzie indziej nazwy zmiennych w kodzie oraz poprawia jego przejrzystość.

Przykład. (Prolog)
Przykładowe wykorzystanie zmiennej oraz zmiennej anonimowej w zapytaniu wygląda następująco:

```
rodzic(X, iwona).    % kto jest rodzicem Iwony?
X = aniela

rodzic(_, iwona).    % czy jakikolwiek rodzic Iwony
Yes                  % jest znany?
```

Komentarze możemy rozumieć jako interpretację zapytania w języku naturalnym.

Należy pamiętać, że wielokrotnym wystąpieniom zmiennej anonimowej w wyrażeniu mogą być przypisane różne wartości.

Przykład. (Prolog)

```
?- rodzic(iwona, ola) = rodzic(X, Y).
X = iwona,
Y = ola
?- rodzic(iwona, ola) = rodzic(X, X).
No
?- rodzic(iwona, ola) = rodzic(_, _).
Yes
```

Strukturą może być *lista* lub *złożony obiekt*. Struktur używamy do grupowania danych.

Przykład. (Prolog)
Przykładowymi strukturami są:

```
[]                    % pusta lista
[1, 2, 3]             % trójelementowa lista
[[a], [b1, b2]]       % lista z zagnieżdżonymi listami
data(16, 6, 2017).    % struktura definiująca datę
                      % (dzień, miesiąc, rok)
```

2.4 Ogólna charakterystyka i definicja pojęcia indukcyjnego programowania w logice

Indukcyjne programowanie logiczne (ang. *inductive logic programming, ILP*) to dział uczenia maszynowego, reprezentujący dane i hipotezy za pomocą logiki predykatów pierwszego rzędu, którego celem jest szukanie wzorców w dostarczonych danych. Uczenie jest realizowane poprzez znajdowanie logicznej reguły wynikającej z relacji określonych w zdefiniowanej wiedzy dziedzinowej.

Definicja 2.34. *Indukcyjne programowanie logiczne* zakłada, że dana jest wiedza dziedzinowa B oraz zbiór przykładów E. Zbiór przykładów definiujemy jako $E = E^+ \cup E^-$, gdzie E^+ oznacza zbiór przykładów pozytywnych, a E^- zbiór przykładów negatywnych. Hipotezą H nazywamy taką formułę logiczną, z której w koniunkcji z B można logicznie wyprowadzić wszystkie przykłady pozytywne i żadnego przykładu negatywnego.

2.4.1 Charakterystyka wejścia i wyjścia

Definicja 2.35. *Wejście* przyjmuje trzy parametry:

E^+ zbiór przykładów pozytywnych,

E^- zbiór przykładów negatywnych,

B wiedza dziedzinowa (ang. *background knowledge*) wyrażona jako zbiór definicji predykatów (patrz: 2.4.2).

Definicja 2.36. *Wyjście* to formuła logiczna H (*hipoteza*) taka, że:

— wszystkie przykłady pozytywne $\in E^+$ można logicznie wyprowadzić z $B \wedge H$ (**pełność**),

— żadnego z przykładów negatywnych $\in E^-$ nie można wyprowadzić z $B \wedge H$ (**spójność**).

2.4.2 Wiedza dziedzinowa

Poprzez wiedzę dziedzinową rozumiemy dostarczoną wiedzę **ekstensjonalną** oraz **intensjonalną**.

Definicja 2.37. *Ekstensjonalna wiedza dziedzinowa* to wiedza określona za pomocą faktów.

Przykład. (Prolog)
Wiedzę ekstensjonalną można przedstawić następująco:

```
kobieta(iwona).
rodzic(iwona, ola).
```

Definicja 2.38. *Intensjonalna wiedza dziedzinowa* to definicje predykatów lub reguły.

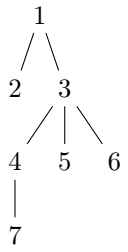
Przykład. (Prolog)
Wiedzę intensjonalną można przedstawić następująco:

```
matka(X, Y) :- kobieta(X), rodzic(X, Y).
babka(X, Y) :- kobieta(X), rodzic(X, Z), rodzic(Z, Y).
```

2.5 Przykład działania metody dedukcji na podstawie relacji rodzinnych

2.5.1 Dane wejściowe i określenie problemu

Na rysunku (2) dane jest drzewo relacji rodzinnych określonych na zbiorze kobiet $V = \{1, 2, 3, 4, 5, 6, 7\}$. Na jego podstawie chcemy zdefiniować relację `babka(X, Y).` taką, że „X jest babką Y".



Rysunek 2: Opis relacji rodzinnych określonych na zbiorze V

Wiedzę dziedzinową B opisujemy z użyciem predykatu `rodzic(X, Y).` takiego, że „X jest rodzicem Y". Dysponujemy zatem następującymi faktami:

```
rodzic(1, 2).
rodzic(1, 3).
rodzic(3, 4).
rodzic(3, 5).
rodzic(3, 6).
rodzic(4, 7).
```

Z rysunku (2) możemy odczytać zbiór przykładów pozytywnych

$$E^+ = \{(1,4), (1,5), (1,6), (3,7)\}$$

dla szukanej relacji bycia babką. Dwuargumentowa relacja `babka/2` między pozostałymi obiektami tworzy zbiór przykładów negatywnych E^-.

2.5.2 Opis algorytmu

Jeśli założymy, że w B jest dostępny tylko predykat `rodzic/2` i ograniczymy się tylko do zmiennych wykorzystywanych w głowie klauzuli, to otrzymamy potencjalne reguły definiujące relację `babka/2`:

```
babka(X, Y) :- rodzic(X, Y).
babka(X, Y) :- rodzic(Y, X).
babka(X, Y) :- rodzic(X, X).
babka(X, Y) :- rodzic(Y, Y).
```

Nie pokrywają one przykładów pozytywnych.

Zmieńmy założenia: pozwólmy na użycie jednego argumentu więcej, takiego, który nie znajduje się w głowie klauzuli. Pozwala to na wprowadzenie czterech nowych literałów wykorzystujących nową zmienną `Z`:

```
rodzic(X, Z), rodzic(Y, Z), rodzic(Z, X), rodzic(Z, Y).
```

Następnie stawiamy hipotezę i oceniamy ją korzystając z podstawień dla trójki `(X, Z, Y)`.

W przykładzie mamy $7^3 = 343$ możliwych realizacji trójki `(X, Z, Y)`. Zgodne z przykładami pozytywnymi z E^+ są tylko:

$$\{(1,3,4),(1,3,5),(1,3,6),(3,4,7)\}$$

Weźmy przykładową klauzulę `babka(X, Y) :- rodzic(X, Z).` Jest ona niespójna, ponieważ w głowie klauzuli występuje zmienna `Y`, która nie pojawia się w ciele reguły. Podobnie klauzule:

```
babka(X, Y) :- rodzic(Y, Z).
babka(X, Y) :- rodzic(Z, X).
babka(X, Y) :- rodzic(Z, Y).
```

także są niespójne.

Dokonujemy dalszej specjalizacji poprzez rozszerzenie definicji predykatu `babka/2` o kolejne wystąpienie predykatu `rodzic/2`. Generujemy pierwszy zestaw dla `babka(X, Y) :- rodzic(X, Z).` i w wyniku otrzymujemy:

```
babka(X, Y) :- rodzic(X, Z), rodzic(X, Z).
babka(X, Y) :- rodzic(X, Z), rodzic(Y, Z).
babka(X, Y) :- rodzic(X, Z), rodzic(Z, X).
babka(X, Y) :- rodzic(X, Z), rodzic(Z, Y).
```

Dla każdego tak wygenerowanego zestawu reguł sprawdzamy pokrycie przykładów pozytywnych i negatywnych.

Zauważamy, że klauzula `babka(X, Y) :- rodzic(X, Z), rodzic(Z, Y).` pokrywa następujące trójki:

$$\{(1,3,4),(1,3,5),(1,3,6),(3,4,7)\}$$

oraz nie pokrywa żadnego przykładu negatywnego z E^-. Możemy zatem pozostać przy powyższej hipotezie.

2.6 Zastosowania

Indukcyjne programowanie w logice (IPL) znajduje zastosowanie w następujących dziedzinach:

Farmacja i bioinformatyka

— predykcja mutageniczności związków chemicznych,

— projektowanie nowych związków chemicznych lub leków (ang. *Structure/Activity Relationships*),

— predykcja struktury białek i ich biologicznej funkcji.

Mechanika i projektowanie inżynierskie

— metoda elementów skończonych (ang. *mesh*),

— analiza sterowania procesami technologicznymi.

Zastosowanie w ochronie środowiska

— klasyfikacja wody w rzekach, predykcja biodegradacji związków chemicznych.

Przetwarzanie języka naturalnego

— automatyczna konstrukcja pareserów języka naturalnego,

— uczenie się past tense czasowników w języku angielskim,

— analiza morfologiczna.

Text-mining / Web-mining

3 Eksperyment wzorcowy

Eksperyment wzorcowy został przedstawiony na podstawie pracy *Inductive Logic Programming With Large-Scale Unstructured Data* [BS94] i odtworzony zgodnie z wytycznymi w niej zawartymi, z użyciem dostarczonych, nieustrukturyzowanych danych wejściowych dla narzędzia GOLEM.

3.1 Opis problemu

Artykuł [BS94] opisuje problem gry końcowej w partii szachowej, gdzie na szachownicy pozostały tylko trzy bierki: biały król, biała wieża oraz czarny król (ang. *King and Rook against King endgame (KRK)*), a kolejny ruch na planszy należy do czarnych (ang. *Black's turn to move (BTM)*). Posiadając bazę danych, która dla zbioru pozycji KRK definiuje optymalną liczbę ruchów do wygranej taką, że: „przy założeniu ruchu czarnych i zadanym układzie KRK białe wygrywają optymalnie w n ruchach" (ang. *black-to-move KRK position won optimally for white in n moves*) można wydedukować reguły dla poszczególnych wartości n.

Taki problem można zamodelować za pomocą predykatu `krk/7`, gdzie `krk` to nazwa, jaką mu nadajemy, a liczba `7` oddzielona od niej ukośnikiem, to liczba argumentów, jakie przyjmuje predykat (arność). Przyjęto, że kolejne argumenty `krk/7` to:

1. liczba ruchów pozostałych do wygranej białych (przy założeniu BTM),

2, 3. pozycja białego króla,

4, 5. pozycja białej wieży,

6, 7. pozycja czarnego króla.

Pozycję, na której znajduje się bierka, oznacza się, podając jako pierwszy argument kolumnę, a drugi — rząd.

Przykład. (Prolog)
`krk(2,c,1,g,8,a,1).` należy rozumieć jako: „zakładając ruch czarnych (BTM), białe wygrywają w 2 ruchach przy położeniu białego króla na c1, białej wieży na g8 i czarnego króla na a1".

3.2 Dostarczone dane

Dla problemu KRK dostarczone zostały dwa pliki źródłowe z następującymi danymi:

(1) `krk.b` — wiedza dziedzinowa na temat problemu,

(2) `krk_win` — baza przykładów pozytywnych.

3.2.1 Wiedza dziedzinowa

Plik (1) zawiera definicje trzech predykatów, które pozwolą zdeterminować docelowy predykat `krk/7`.

Kod źródłowy 1 ./../experiment/chess–model/krk_win/krk.b

```
11: num(7).
12: num(8).
13: num(a).
14: num(b).
15: num(c).
```

Powyższy jednoargumentowy predykat `num/1` definiuje oznaczenia kolumn oraz rzędów.

(12) należy rozumieć jako: „`8` jest oznaczeniem kolumny lub rzędu”,

(13) należy rozumieć jako: „`a` jest oznaczeniem kolumny lub rzędu”.

Do oznaczenia krawędzi szachownicy — tj. skrajnych kolumn i rzędów — zdefiniowany został predykat `edge/1`.

Kod źródłowy 2 ./../experiment/chess–model/krk_win/krk.b

```
22: edge(1).
23: edge(8).
24: edge(a).
25: edge(h).
```

Zatem:

(23) należy rozumieć jako: „`8` jest skrajną kolumną lub rzędem szachownicy”,

(24) należy rozumieć jako: „`a` jest skrajną kolumną lub rzędem szachownicy”.

Za pomocą wymienionych predykatów `num/1` i `edge/1` można oznaczyć pola szachownicy. Aby określić ich relację pomiędzy sobą należy użyć trójparametrowego predykatu `diff/3`.

Kod źródłowy 3 ./../experiment/chess–model/krk_win/krk.b

```
27: diff(1,1,d0).
28: diff(2,2,d0).
29: diff(3,3,d0).
30: diff(4,4,d0).
31: diff(5,5,d0).
32: diff(6,6,d0).
33: diff(7,7,d0).
34: diff(8,8,d0).
35:
36: diff(2,1,d1).
```

```
37: diff(3,1,d2).
38: diff(3,2,d1).
39: diff(4,1,d3).
40: diff(4,2,d2).
```

Predykat jako dwa pierwsze parametry przyjmuje oznaczenia rzędów i kolumn. Trzecim parametrem jest odległość pomiędzy nimi. Należy określić wszystkie odległości — także zerowe — pomiędzy kolumnami między sobą, rzędami między sobą oraz pomiędzy kolumnami i rzędami w następujący sposób:

(27) należy rozumieć jako: „odległość pomiędzy kolumną `1` oraz `1` jest równa 0”,

(28) należy rozumieć jako: „odległość pomiędzy kolumną `2` oraz `2` jest równa 0”, itd.

oraz:

(36) należy rozumieć jako: „odległość pomiędzy kolumną `2` oraz `1` jest równa 1”,

(37) należy rozumieć jako: „odległość pomiędzy kolumną `3` oraz `1` jest równa 2”, itd.

Pliki z rozszerzeniem `*.b` są automatycznie interpretowane przez narzędzie GOLEM jako zawierające wiedzę dziedzinową (ang. *background knowledge*).

3.2.2 Przykłady pozytywne

Plik (2) zawiera bazę przykładów pozytywnych dla wygranej w n ruchach (dla $n = 0, \cdots, 16$, gdzie 0 oznacza natychmiastową wygraną) oraz dla rozgrywki zakończonej remisem ($n = -1$).

Lista przykładów jest posortowana po n i dla każdego n rozpoczyna się linią komentarza z oznaczeniem ilości ruchów do wygranej, jak w zaprezentowanych poniżej fragmentach pliku (2), w liniach nr 1, 29, 1108 oraz 25278. Liczbę ruchów do wygranej białych przy założeniu ruchu czarnych (BTM) oznaczoną jako n nazywa się *głębokością* (ang. *depth*).

Kod źródłowy 4 ./../experiment/chess–model/krk_win/krk_win

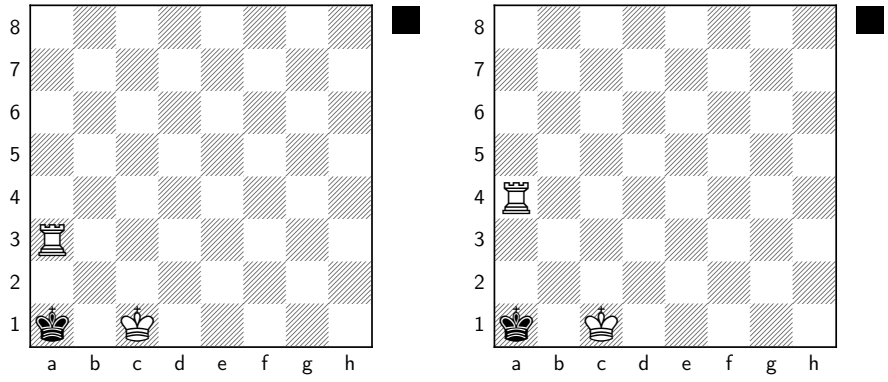
```
1: % BTM depth 00
2: krk(0,c,1,a,3,a,1).
3: krk(0,c,1,a,4,a,1).
```

Powyższy kod można zinterpretować następująco:

(1) komentarz rozpoczynający podzbiór przykładów pozytywnych dla głębokości 0,

(2) przedstawia się na szachownicy jak na poniższym rysunku (3a) i rozumie jako: „zakładając ruch czarnych (BTM) oraz przy położeniu białego króla na c1, białej wieży na a3 i czarnego króla na a1, białe wygrywają w 0 ruchach (natychmiast)”,

(3) przedstawia się na szachownicy jak na poniższym rysunku (3b) i rozumie jako: „zakładając ruch czarnych (BTM) oraz przy położeniu białego króla na c1, białej wieży na a4 i czarnego króla na a1, białe wygrywają w 0 ruchach (natychmiast)”.



(a) krk_win:2 (b) krk_win:3

Rysunek 3: Graficzne przedstawienie na szachownicy dostarczonych w pliku (2) przykładów pozytywnych

Analogicznie można zinterpretować kolejny podzbiór:

Kod źródłowy 5 ./../experiment/chess–model/krk_win/krk_win

```
29: % BTM depth 01
30: krk(1,c,1,c,3,a,2).
31: krk(1,c,1,d,3,a,2).
```

(29) komentarz rozpoczynający podzbiór przykładów pozytywnych dla głębokości 1,

(30) należy rozumieć jako: „zakładając ruch czarnych (BTM) oraz przy położeniu białego króla na c1, białej wieży na c3 i czarnego króla na a2, białe wygrywają w 1 ruchu”,

(31) należy rozumieć jako: „zakładając ruch czarnych (BTM) oraz przy położeniu białego króla na c1, białej wieży na d3 i czarnego króla na a2, białe wygrywają w 1 ruchu”.

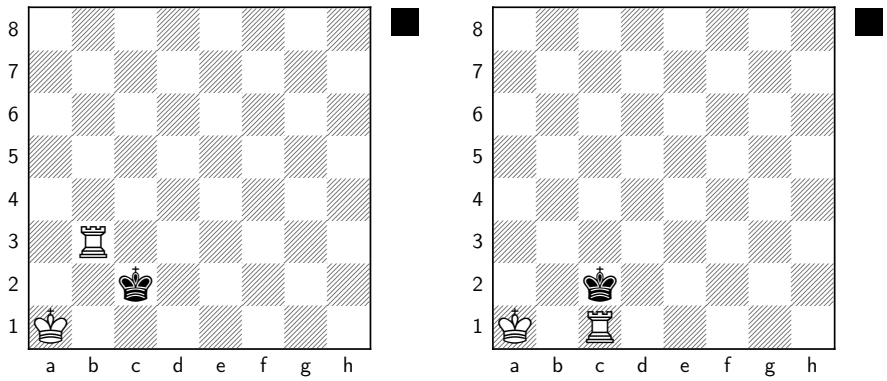
W zaprezentowany sposób są dostarczone przykłady aż do głębokości równej 16 oraz dla remisu, który jest oznaczony jako $n = -1$.

Kod źródłowy 6 ./../experiment/chess–model/krk_win/krk_win

```
25276: krk(16,b,1,g,7,f,5).
25277: krk(16,b,1,g,7,g,5).
25278: % BTM drawn
25279: krk(-1,a,1,b,3,c,2).
25280: krk(-1,a,1,c,1,c,2).
```

(25279) przedstawia się na szachownicy jak na poniższym rysunku (4a) i rozumie jako: „zakładając ruch czarnych (BTM) oraz przy położeniu białego króla na a1, białej wieży na b3 i czarnego króla na c2, gra kończy się remisem",

(25280) przedstawia się na szachownicy jak na poniższym rysunku (4b) i rozumie jako: „zakładając ruch czarnych (BTM) oraz przy położeniu białego króla na a1, białej wieży na c1 i czarnego króla na c2, gra kończy się remisem".



(a) krk_win:25279 (b) krk_win:25280

Rysunek 4: Graficzne przedstawienie na szachownicy dostarczonych w pliku (2) przykładów pozytywnych

3.3 Przebieg eksperymentu

Opisany w [BS94] problem został podzielony na zbiór mniejszych, możliwych do scalenia problemów w następujący sposób: każda głębokość stała się osobnym problemem, który rozwiązuje się niezależnie. Tak więc dla każdej głębokości ($n = -1, 0, \cdots, 16$) należy wyodrębnić zestaw danych potrzebnych do wydedukowania reguł za pomocą narzędzia GOLEM.

Dla wygenerowania poprawnego wejścia dla narzędzia GOLEM potrzebna jest wiedza dziedzinowa, zestaw przykładów pozytywnych i negatywnych.

Wiedza dziedzinowa dla każdego zestawu będzie tym samym, kompletnym plikiem (1) dostarczonym z modelowym rozwiązaniem.

Pliki z przykładami pozytywnymi generuje się wyodrębniając z pliku (2) tylko linie zawierające przykłady pozytywne dla zadanej głębokości. Dla $n = 0$ są to linie od (1) do (28), dla $n = 1$ — linie od (29) do (107), itd.

Pliki z przykładami negatywnymi generuje się natomiast spośród pozostałych przykładów pozytywnych dla każdej zadanej głębokości, np. dla $n = 2$ zbiór przykładów negatywnych jest losowany spomiędzy przykładów pozytywnych dla głębokości innych niż równa 2, tj. ze zbioru linii od (1) do (107) oraz od linii (335) do końca pliku (2). Liczba losowanych przykładów w eksperymencie modelowym opisanym w [BS94] to maksymalnie 1000.

Tak przygotowane dane pozwalają indukować predykat `krk/7` za pomocą narzędzia GOLEM dla każdej z zadanych głębokości.

Ostatnim krokiem jest scalenie otrzymanych rozwiązań częściowych.

4 Cele oraz przeprowadzony eksperyment

Celem pracy jest odtworzenie i rozszerzenie istniejącego eksperymentu przeprowadzonego w artykule [BS94]. Eksperyment ma zostać odtworzony w sposób zautomatyzowany tak, aby móc łatwo generować wymagane pliki wejściowe dla narzędzia GOLEM w wybranym zakresie — tj. dla zadanego n, gdzie $n = -1,\ 0,\ \cdots,\ 16$ i oznacza głębokość.

Interesującym rozszerzeniem modelowego eksperymentu jest wskazanie ścieżki prowadzącej do wygranej białych bierek. Osiągnięcie tego celu jest możliwe przy pomocy dostarczonych danych: wiedzę dziedzinową i scalone wyjście narzędzia GOLEM dla każdej zadanej głębokości rozszerza się o klauzule definiujące możliwe przejścia pomiędzy dostarczonymi zestawami pozycji dla gry końcowej KRK. Nowe klauzule budowane są z wykorzystaniem wiedzy dziedzinowej dołączonej do eksperymentu modelowego.

4.1 Odtworzenie eksperymentu modelowego

4.1.1 Potrzebne dane

Dla poprawnego wygenerowania wyjścia przez narzędzie GOLEM potrzebny jest kompletu plików dla każdego zestawu rozpatrywanych głębokości:

`*.b` — plik zawierający wiedzę dziedzinową (ang. ***b****ackground*),

`*.f` — plik zawierający przykłady pozytywne (ang. ***f****oreground*),

`*.n` — plik zawierający przykłady negatywne (ang. ***n****egative*).

Plik z wiedzą dziedzinową `krk.b` jest dostarczony z danymi eksperymentu modelowego i wykorzystany bez wprowadzania zmian.

Plik `krk_win` z przykładami pozytywnymi został dostarczony z rozwiązaniem modelowym w postaci opisanej jak w rozdziale 3.2.2. i należy podzielić go według wytycznych z rozdziału 3.3.

Wyjściem narzędzia GOLEM są pliki z rozszerzeniem `*.r` zawierające reguły (ang. ***r****ules*) dla każdego zestawu rozpatrywanych głębokości, które należy scalić w jedno rozwiązanie.

4.1.2 Analiza kodu źródłowego

Wynikiem poniższego skryptu bash jest jedno, scalone rozwiązanie problemu, uwzględniające zadany przedział głębokości $0 \leq n \leq 16$. W rozwiązaniu dla uproszczenia nie rozpatrujemy remisów.

Skrypt przyjmuje jeden obowiązkowy parametr, którym jest głębokość.

Kod źródłowy 7 ./../experiment/chess–carried/generate–n.sh

```
1: #!/bin/bash
2:
3: DEPTH=$1
```

```
 4: PATTERN='% BTM depth '
 5: INPUT='./../chess-model/krk_win'
 6: TEMP='./tmp'
 7: OUTPUT='.'
 8: GOLEM='./../golem/src/golem'
 9: LASTLINE=1
10: mkdir -p "$TEMP"
11: "" > "$OUTPUT/krk.r"
12: cp "$INPUT/krk.b" "$OUTPUT/krk.b"
13: cp "$INPUT/krk.b" "$TEMP/krk0.b"
14: sed -i 's/krk\/7/krk\/6/g' "$TEMP/krk0.b"
15:
16: for (( d=0; d<=DEPTH; )); do
17:   if [ $d -gt 0 ]; then
18:     cp "$TEMP/krk0.b" "$TEMP/krk$d.b"
19:   fi
20:
21:   FILENAME="$TEMP/krk$d"
22:   ((d+=1))
23:   p=$PATTERN
24:   if [ $d -lt 10 ]; then
25:     p+='0'
26:   fi
27:   p+=$d
28:
29:   LINE=$(grep -n -m 1 "$p" "$INPUT/krk_win" | cut -d: -
          f1)
30:   tail -n +$LINE "$INPUT/krk_win" | shuf -n 100 | sed -
          e 's/krk(-\?[0-9]\+,/krk(/g' > "$FILENAME.n"
31:   head -n $LINE "$INPUT/krk_win" | tail -n +$LASTLINE |
           sed -e 's/krk(-\?[0-9]\+,/krk(/g' > "$FILENAME.f
          "
32:   $GOLEM $FILENAME
33:   LASTLINE=$LINE
34:
35:   cat "$FILENAME.r" | sed -e "s/krk(/krk($((d-1)),/g"
          >> "$OUTPUT/krk.r"
36: done
```

W liniach (3)–(9) definiujemy zmienne, które ułatwią generowanie nazw plików i przebieg skryptu. Kolejno od linii (3):

(3) `DEPTH` — głębokość, której przypisujemy wartość podaną jako parametr podczas uruchomienia skryptu,

(4) `PATTERN` — wzór komentarza oddzielającego zestawy różnych głębokości,

(5) `INPUT` — ścieżka katalogu zawierającego pliki dostarczone z eksperymentem modelowym,

(6) `TEMP` — ścieżka katalogu tymczasowego, w którym zostaną zapisane wygenerowane pliki dla narzędzia GOLEM (`*.b`, `*.f`, `*.n`) oraz jego wyjście (`*.r`) dla każdej zadanej głębokości,

(7) `OUTPUT` — katalog, w którym zostaną zapisane pliki z gotowym, scalonym rozwiązaniem problemu dla zadanego przedziału głębokości $0 \leq n \leq 16$,

(8) `GOLEM` — ścieżka do narzędzia GOLEM,

(9) `LASTLINE` — zmienna pomocnicza, przechowująca ostatnią linię przetworzonego zestawu dla danej głębokości.

Następnym krokiem skryptu jest utworzenie kopii plików eksperymentu modelowego, na których będzie można pracować oraz stworzenie (jeśli nie istnieje) struktury katalogów dla rozwiązań częściowych i rozwiązania scalonego.

W linii (10) tworzony jest katalog tymczasowy dla rozwiązań częściowych za pomocą `mkdir`. Przełącznik `-p` sprawia, że program wygeneruje katalogi dla całej ścieżki podanej w argumencie, a w przypadku, gdy ścieżka (lub jej część) istnieje, nie przerwie działania.

W linii (11) przekierowuje się pusty ciąg znaków do pliku `krk.r`, który będzie zawierał scalone rozwiązanie problemu. Przekierowanie ma na celu zapewnienie, że plik:

— istnieje,

— jest pusty.

W dalszej części skryptu są do niego dopisywane częściowe rozwiązania.

Linie (12)–(13) kopiują plik `krk.b` z wiedzą dziedzinową do katalogu ze scalonym rozwiązaniem oraz do katalogu tymczasowego, z nazwą zmienioną odpowiednio dla zerowego zestawu danych. Plik w katalogu z rozwiązaniem scalonym nie jest modyfikowany. Jest on skopiowany tylko w celu ułatwienia późniejszego wczytania otrzymanych w eksperymencie danych do programu SWI-Prolog z poziomu tego katalogu.

Linia (14) przygotowuje kopię wiedzy dziedzinowej do dalszego przetwarzania. Jako, że problem jest rozbity na mniejsze części poprzez rozpatrywanie każdej z zadanych głębokości z osobna, to należy wykluczyć głębokość z predykatu `krk/7`. Oznacza to zmianę definicji predykatu na `krk/6`. Używając programu `sed` i korzystając z wyrażeń regularnych ciąg znaków `krk/7` zostaje podmieniony na `krk/6` w całym pliku (flaga `g` (ang. ***g**lobal*) na końcu wyrażenia).

Linia (16) rozpoczyna pętlę iterującą po głębokości $0 \leq d \leq n$ dla zadanego n równego wartości parametru przekazanego do programu. Pętla nie ma określonego kroku — zostanie to wyjaśnione w kolejnych akapitach — i kończy się w ostatniej linii skryptu (36).

Linie (17)–(19) wykonują kopię przygotowanego wcześniej pliku z wiedzą dziedzinową dla problemu, opisanego w rozdziale 3.2.1. Plik jest kopiowany podczas każdej iteracji (poza zerową) ze względów praktycznych i nie jest dodatkowo modyfikowany. Każdy zestaw danych wczytuje dokładnie taki sam plik `*.b`.

Linia (21) tworzy zmienną `FILENAME`, która przechowuje ścieżkę dla plików tymczasowych dla wejścia narzędzia GOLEM. Zmienna `FILENAME` jest różna w każdym przebiegu pętli, ponieważ dotyczy zawsze innego zestawu głębokości (jest zależna od zmiennej `$d`).

Linia (22) inkrementuje zmienną sterującą pętlą. Inkrementacja następuje w tym miejscu, ponieważ w dalszej części skryptu nie będziemy potrzebować wartości aktualnego przebiegu — chcemy za to wygenerować ze wzoru `PATTERN` treść linii komentarza rozpoczynającego kolejny zestaw danych, który równocześnie kończy aktualnie rozpatrywany.

Treść komentarza rozdzielającego zestawy danych jest generowana w liniach (23)–(27), z uwzględnieniem wiodącego zera przy numeracji głębokości w instrukcji sterującej `if` w linii (24). Jego treść zapisujemy w zmiennej `p`.

Zmienna `LINE` w linii (29) jest inicjowana numerem linii, w której kończy się rozpatrywany zestaw (znajduje się komentarz rozdzielający). Program `grep` wyszukuje zadany ciąg `p`. Przełączniki oznaczają, że program:

— `-n`, `--line-number` — wyświetla linię z dopasowanym ciągiem poprzedzoną numerem linii w pliku,

— `-m 1`, `--max-count=1` — odnajduje maksymalnie 1 wynik.

Wygenerowane przez komendę `grep` wyjście zostaje połączone z wejściem programu `cut` poprzez potok procesowy (ang. *pipe*, ozn. `|`). Program `cut` zwraca wybrane fragmenty dla każdej linii dostarczonego wejścia. Użyte przełączniki oznaczają, że program:

— `-d:`, `--delimiter=:` — definiuje znak dwukropka (`:`) jako separator,

— `-f1`, `--fields=1` — definiuje listę pól, które należy zwrócić — w tym przypadku: jednoelementowa lista zawierająca pole o indeksie 1.

Wynikiem jest numer linii, wycięty z łańcucha znaków.

Przykład. (bash)
Przykład działania programów `grep` i `cut` dla głębokości równej 0:

```
$ grep -n -m 1 "%␣BTM␣depth␣00" "./../chess-model/krk_win/
    krk_win"
1:% BTM depth 00
$ grep -n -m 1 "%␣BTM␣depth␣00" ./../chess-model/krk_win/
    krk_win | cut -d: -f1
1
```

W linii (30) jest losowany zestaw przykładów negatywnych. Wybierane są spośród przykładów pozytywnych dla kolejnych zestawów. Do wyznaczenia części pliku `krk_win`, z której będą losowane przykłady, używa się programu `tail`. Program domyślnie zwraca ostatnie 10 linii wskazanego pliku. Użyty przełącznik `-n +$LINE` (`--lines=+$LINE`) oznacza, że zwrócona zostanie część pliku zaczynająca się od linii `$LINE` do końca (ang. *end of file (EOF)*). Otrzymany wynik jest

przekazany przez potok do programu `shuf`, który generuje losową permutację i zwraca jej 100 pierwszych linii (przełącznik `-n 100`, `--head-count=100`). Wynik jest dalej przekazany poprzez potok do programu `sed` i konsekwentnie, tak jak w linii (14), za pomocą wyrażenia regularnego zostaje zmieniona arność predykatu `krk` poprzez usunięcie pierwszego parametru (głębokości). Tak otrzymany wynik potoku zostaje przekierowany do pliku z rozszerzeniem `*.n`.

Przykłady pozytywne są przetwarzane analogicznie do negatywnych. Potok procesowy z linii (31) rozpoczyna się od programu `head`. Program domyślnie zwraca 10 pierwszych linii wskazanego pliku. Przełącznik `-n $LINE` (`--lines= $LINE`) drukuje pierwsze `$LINE` linii pliku. Łącząc przez potok otrzymane wyjście z wejściem programu `tail` otrzymana zostaje środkowa część pliku `krk_win`, która zawiera tylko zestaw o rozpatrywanej głębokości `$d`. Dalej za pomocą programu `sed` zostaje skorygowana liczba parametrów predykatu `krk`. W efekcie cały potok zostaje przekierowany do pliku z rozszerzeniem `*.f`.

W linii (32) następuje uruchomienie narzędzia GOLEM. Jako parametr wywołania należy podać nazwę zestawu plików (`*.b`, `*.f`, `*.n`).

Zmiennej `LASTLINE` w linii (33) zostaje przypisana wartość `$LINE`. `$LASTLINE` w każdym przebiegu pętli przechowuje numer pierwszej linii rozpatrywanego zestawu głębokości `$d` (numer ostatniej linii z poprzedniego przebiegu pętli).

W linii (35) wynik działania narzędzia GOLEM (wygenerowany plik z rozszerzeniem `*.r`) jest przetwarzany w potoku programem `sed` — z użyciem wyrażenia regularnego dodany zostaje pierwszy parametr predykatu `krk`. Wynik jest dopisywany do pliku `krk.r` z rozwiązaniem całościowym problemu.

4.2 Rozszerzenie eksperymentu modelowego

4.2.1 Opis

Eksperyment zostaje rozszerzony o skrypt napisany w języku Prolog. Zawiera on definicje predykatów `moveKing/4`, `moveRook/4` oraz `find/8`.

Predykaty `moveKing/4` i `moveRook/4` są traktowane jako pomocnicze i zdefiniowane z użyciem dostarczonej wiedzy dziedzinowej eksperymentu modelowego (plik `krk.b`). Opisują one możliwe, dozwolone w rozgrywce szachowej ruchy figur — kolejno: króla i wieży.

Predykat `find/8` znajduje ścieżkę dla wygranej białych bierek na podstawie reguł wyprodukowanych przez narzędzie GOLEM (plik `krk.r`) oraz opisanych w poprzednim akapicie predykatów pomocniczych. Predykat `find/8` jest rekurencyjny.

4.2.2 Analiza kodu źródłowego

Poniższy skrypt jest zapisany jako plik wykonywalny i automatycznie uruchamiamy za pomocą programu SWI-Prolog, co oznacza zapis w linii (1).

Kod źródłowy 8 ./../experiment/chess–carried/find.pl

```
1: #!/usr/bin/swipl
```

```
 2:
 3: :- include('krk.b').
 4: :- include('krk.r').
 5:
 6: moveKing(C, R, NC, NR) :- diff(C, NC, d1), diff(R, NR,
      d1).
 7: moveKing(C, R, NC, NR) :- diff(C, NC, d0), diff(R, NR,
      d1).
 8: moveKing(C, R, NC, NR) :- diff(C, NC, d1), diff(R, NR,
      d0).
 9: moveRook(C, R, NC, NR) :- diff(C, NC, d0), diff(R, NR,
      D), not(D = d0).
10: moveRook(C, R, NC, NR) :- diff(C, NC, D), not(D = d0),
      diff(R, NR, d0).
11:
12: find(WKc, WKr, WRc, WRr, BKc, BKr, [[0, WKc, WKr, WRc,
      WRr, BKc, BKr]], 0) :- krk(0, WKc, WKr, WRc, WRr,
      BKc, BKr).
13: find(WKc, WKr, WRc, WRr, BKc, BKr, [[Moves,  WKc, WKr,
      WRc, WRr, BKc, BKr] | Rest], Moves) :-
14:   krk(Moves, WKc, WKr, WRc, WRr, BKc, BKr),
15:   moveKing(WKc, WKr, WKc2, WKr2),
16:   moveKing(BKc, BKr, BKc2, BKr2),
17:   MovesX is Moves - 1,
18:   find(WKc2, WKr2, WRc, WRr, BKc2, BKr2, Rest, MovesX).
19: find(WKc, WKr, WRc, WRr, BKc, BKr, [[Moves, WKc, WKr,
      WRc, WRr, BKc, BKr] | Rest], Moves) :-
20:   krk(Moves, WKc, WKr, WRc, WRr, BKc, BKr),
21:   moveRook(WRc, WRr, WRc2, WRr2),
22:   moveKing(BKc, BKr, BKc2, BKr2),
23:   MovesX is Moves - 1,
24:   find(WKc, WKr, WRc2, WRr2, BKc2, BKr2, Rest, MovesX).
```

W liniach (3) i (4) załączone zostają:

(3) plik `krk.b` zawierający wiedzę dziedzinową,

(4) plik `krk.r` będący wynikiem działania narzędzia GOLEM, wygenerowany przez skrypt opisany w rozdziale 4.1.2.

Predykat `moveKing/4` jest opisany w liniach od (6) do (8). Przyjmuje on cztery parametry związane z wykonaniem ruchu figurą króla:

`C` kolumnę początkowego położenia króla (ang. ***c**olumn*),

`R` rząd początkowego położenia króla (ang. ***r**ow*),

`NC` kolumnę docelowego położenia króla (ang. ***n**ew **c**olumn*),

`NR` rząd docelowego położenia króla (ang. ***n**ew **r**ow*).

Figura króla może poruszać się o jedno pole w każdym kierunku. Przykładowe dozwolone ruchy przedstawia znajdujący się na stronie 40 rysunek 5a. Zgodnie z przedstawionym rysunkiem można stwierdzić, że należy rozpatrzyć trzy przypadki:

(6) ruch po skosie; zmienia się kolumna i wiersz,

(7) ruch w obrębie wiersza; zmienia się tylko wiersz,

(8) ruch w obrębie kolumny; zmienia się tylko kolumna.

Gwarancję, że figura poruszy się w danym kierunku tylko o jedno pole (lub nie poruszy się wcale) daje predykat `diff/3`, który jest dostarczony jako część wiedzy dziedzinowej załączonej w linii (3).

Analogicznie jest opisana możliwość wykonania ruchu przez wieżę. Predykat `moveRook/4` przyjmuje argumenty identycznie oznaczone i w takiej samej kolejności co `moveKing/4`:

`C` kolumnę początkowego położenia wieży,

`R` rząd początkowego położenia wieży,

`NC` kolumnę docelowego położenia wieży,

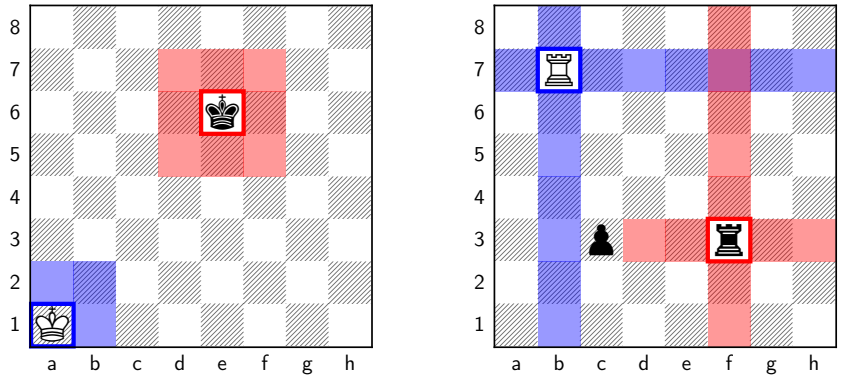
`NR` rząd docelowego położenia wieży.

Figura wieży może poruszać się w wybranym kierunku w obrębie albo rzędu, albo kolumny. Zawsze o dowolną liczbę niezajętych pól. Przykładowe dozwolone ruchy wieży obrazuje poniższy rysunek 5b. Można z niego odczytać dwa przypadki:

(9) ruch w obrębie wiersza; zmienia się tylko wiersz,

(10) ruch w obrębie kolumny; zmienia się tylko kolumna.

Dla uproszczenia predykat `moveRook/4` nie uwzględnia zajętości pól.



(a) Król (b) Wieża

Rysunek 5: Graficzne przedstawienie na szachownicy możliwych, dozwolonych ruchów wybranych figur

Od linii (12) do końca pliku w linii (24) skrypt opisuje predykat `find/8`. Parametry jakie przyjmuje, to:

`WKc, WKr` pozycję białego króla na planszy — kolejno kolumnę i rząd,

`WRc, WRr` pozycję białej wieży na planszy – kolejno kolumnę i rząd,

`BKc, BKr` pozycję czarnego króla na planszy – kolejno kolumnę i rząd,

`Path` przebieg gry końcowej KRK przedstawiony w postaci listy z zagnieżdżonymi listami zawierającymi liczbę ruchów do wygranej białych i położenia wszystkich figur (lista parametrów, które przyjmuje predykat `krk/7`),

`Moves` liczba ruchów do wygranej białych przy zadanym położeniu figur.

Predykat korzysta z wiedzy dziedzinowej i reguł wygenerowanych w ramach odtworzenia eksperymentu modelowego (rozdział 4.1). Jest też skonstruowany w sposób rekurencyjny. Warunkiem zatrzymania rekurencji jest rozgrywka wygrana dla białych bierek (głębokość $n = 0$). Taką sytuację przedstawia linia (12).

W przypadku, gdy liczba ruchów pozostałych do wygranej białych bierek jest większa od zera (`Moves > 0`) w kolejnym kroku należy odnaleźć takie ułożenie bierek na planszy, które:

— jest prawidłowym ułożeniem bierek na planszy, zgodnym z zasadami gry w szachy,

— wynika bezpośrednio z aktualnego położenia figur KRK i ruchów dozwolonych podczas gry w szachy,

— zmniejsza o 1 liczbę ruchów białych bierek do wygranej (`Moves - 1`).

W związku z tym trzeba rozpatrzyć dwa przypadki:

— ruch wykonuje czarny król i biały król,

— ruch wykonuje czarny król i biała wieża.

Pierwszy wariant opisuje reguła z linii (13)–(18), a drugi (19)–(24).

W obu przypadkach schemat postępowania jest identyczny. Najpierw należy odnaleźć liczbę ruchów, w których wygrywają białe bierki przy zadanej pozycji KRK na planszy. Wykorzystany jest do tego predykat `krk/7` — linie (14) i (20). Jeśli taka pozycja istnieje — zgodnie ze stanem wiedzy, tj. w zależności od przyjętej głębokości $0 \leq n \leq 16$, dla której wygenerowany został zbiór reguł w pliku `krk.r` — to w następnym kroku należy odszukać taki układ KRK na planszy, do którego można bezpośrednio dotrzeć wykonując ruch jedną z figur białych: królem — jak w linii (15), lub wieżą — w linii (21), i czarnym królem — jak w linii (16) i (22). Nowy układ bierek musi być poprawny w rozumieniu zasad gry w szachy, co jest zagwarantowane przez predykat `krk/7`. Gwarancją doboru optymalnej rozgrywki gry końcowej KRK jest wybór układu figur, gdzie liczba ruchów białych bierek do wygranej jest mniejsza o 1 — jak w liniach (17) i (23).

Proces należy powtarzać, dopóki nie osiągnie się wygranej — odzwierciedlają to wywołania rekurencyjne predykatu `find/8` w liniach (18) i (24).

4.3 Wizualizacja rozwiązania

Wizualizacja eksperymentu ma postać aplikacji internetowej i wykorzystuje technologie HTML/CSS, JavaScript oraz PHP.

Aplikacja korzysta ze skryptu opisanego w rozdziale 4.2. Skrypt jest wczytywany do programu `swipl` razem z zapytaniem za pomocą funkcji `exec()`.

Kod źródłowy 9 ./../experiment/visualization/chessboard.php

```
10: $cmd = sprintf("swipl␣-q␣-f␣./../chess-carried/find.pl␣
        -g␣'findall([P],␣find(%s,P,M),␣Paths),print(Paths),
        halt'", implode(',', $find));
11:
12: exec($cmd, $output);
```

Wynik zwrócony przez wywołanie funkcji w linii (12) jest następnie parsowany i wyświetlony w postaci listy możliwych rozwiązań z odnośnikiem do interaktywnej reprezentacji na szachownicy.

Uruchomienie wizualizacji na maszynie lokalnej odbywa się z linii poleceń za pomocą wbudowanego serwera programu `php`:

```
$ php -S localhost:8080 chessboard.php
```

z poziomu katalogu zawierającego pliki wizualizacji.

Dodatkowo wizualizacja projektu jest umieszczona na publicznie dostępnym serwerze. Adres URL znajduje się w pliku `README.md` w repozytorium systemu kontroli wersji GIT pod adresem podanym w dodatku A.2.

5 Podsumowanie

5.1 Otrzymane wyniki

Wyniki otrzymane przez odtworzone rozwiązanie modelowe oraz rozszerzający je eksperyment autorski są w znacznym stopniu poprawne, ale też stwarzają pole do dalszego rozwoju. Ze względu na przyjętą taktykę losowania przykładów negatywnych, eksperyment nie jest idealnie powtarzalny — wyniki dla dwóch różnych, wygenerowanych zestawów nie są jednakowe już na poziomie reguł wydedukowanych przez narzędzie GOLEM. Część eksperymentu odtwarzająca rozwiązanie modelowe została zaimplementowana w sposób zautomatyzowany, co pozwoliło zaobserwować i prześledzić ten problem.

5.2 Wnioski

W pracy udało się odtworzyć eksperyment modelowy w zadowalającym stopniu. Został on dla uproszczenia nieznacznie ograniczony o czym można przeczytać w podrozdziale 5.3.

Rozszerzenie rozwiązania modelowego, które zakładało zautomatyzowanie procesu generowania wejście dla narzędzia GOLEM oraz odnalezienie przebiegu gry końcowej KRK, powiodło się i zostało osiągnięte z wykorzystaniem dostarczonej z eksperymentem modelowym wiedzy dziedzinowej w sposób zwięzły i elegancki.

Wizualizacja eksperymentu pozwala prześledzić i ocenić wyniki działania rozwiązania autorskiego oraz odtworzonego eksperymentu modelowego.

5.3 Perspektywy dalszego rozwoju

Projekt przedstawiony w pracy można uzupełnić o kilka problematycznych aspektów.

Pierwszym z nich jest generowanie przykładów negatywnych. Dla uproszczenia w odtworzonym eksperymencie modelowym przykłady negatywne dla głębokości n są losowane ze zbioru przykładów pozytywnych tylko dla głębokości większej od n.

Generator zestawów nie rozpatruje przypadków, gdy rozgrywka szachowa kończy się remisem. W przypadku uwzględnienia takich przykładów należałoby także uzupełnić część rozszerzającą rozwiązanie modelowe.

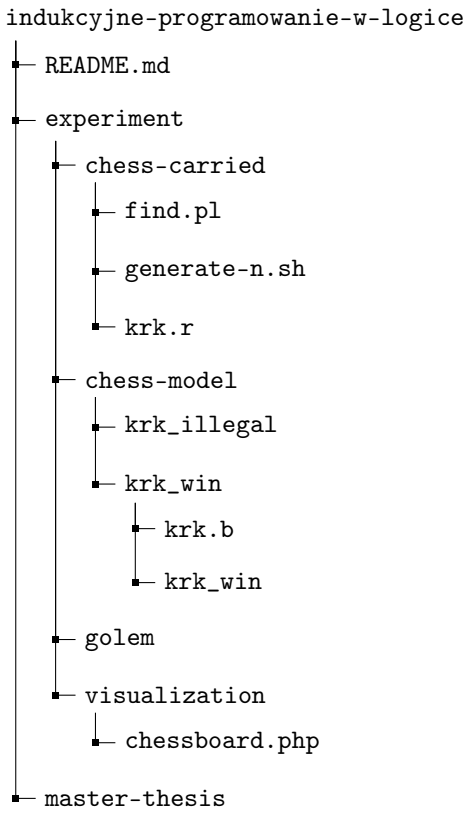
W rozszerzeniu eksperymentu modelowego ruchy wieży nie uwzględniają zajętości pól. Istnieje jednak możliwość uszczegółowienia predykatu odpowiedzialnego za poruszanie się figury wieży po planszy.

Dodatek A Struktura projektu

Poniższy diagram przedstawia schemat struktury katalogów z uwzględnieniem najważniejszych plików opisywanych w pracy. Odnalezienie fragmentów dotyczących poszczególnych plików ułatwi indeks umieszczony na stronie 47.

Ścieżki do cytowanych w pracy plików są ścieżkami relatywnymi. Treść niniejszej pracy znajduje się w katalogu `master-thesis` w repozytorium wymienionym w sekcji A.2.

A.1 Schemat struktury katalogów



A.2 Repozytorium

Pliki projektowe oraz pliki LaTeX tej pracy są dostępne w repozytorium systemu kontroli wersji GIT pod adresem:

`https://bitbucket.org/awichert/indukcyjne-programowanie-w-logice`

Indeks odniesień do plików

Indeks pojęć

Wykaz akronimów

BTM Black's turn to move. 27, 29–31

CSS Cascading Style Sheets. 41

EOF end of file. 36

FOL first-order logic. 14

HTML HyperText Markup Language. 41

IPL indukcyjne programowanie w logice. 5, 9, 25

JS JavaScript. 41

KRK King and Rook against King endgame. 5, 9, 27, 33, 40, 41, 43

KRP klasyczny rachunek predykatów. 7, 14, 15

KRZ klasyczny rachunek zdań. 7, 11–14, 16

PHP PHP: Hypertext Preprocessor. 41

Literatura

[BS94] Michael Bain and Ashwin Srinivasan. Inductive Logic Programming With Large-Scale Unstructured Data. *Machine Intelligence*, 14:233–267, 8 1994.

[Busa] Wojciech Buszkowski. Materiały dydaktyczne dla przedmiotu Elementy logiki I. `http://www.staff.amu.edu.pl/~buszko/logpodinn.pdf`. [Online, 22-06-2017].

[Busb] Wojciech Buszkowski. Materiały dydaktyczne dla przedmiotu Elementy logiki II. `http://www.staff.amu.edu.pl/~buszko/logpodinn.pdf`. [Online, 22-06-2017].

[Busc] Wojciech Buszkowski. Materiały dydaktyczne dla przedmiotu Logiczne podstawy informatyki. `http://www.staff.amu.edu.pl/~buszko/logpodinn.pdf`. [Online, 22-06-2017].

[CM03a] William Clocksin and Chris Mellish. *Programming in Prolog: using the ISO standard*. Springer, 5 edition, 6 2003.

[CM03b] William Clocksin and Chris Mellish. *Prolog. Programowanie*. Helion, 5 2003.

[ISO95] Information technology – programming languages – prolog – part 1: General core. Standard ISO/IEC 13211-1:1995, International Organization for Standardization, Geneva, 6 1995.

[MdR94] Stephen Muggleton and Luc de Raedt. Inductive Logic Programming: Theory And Methods. *Journal of Logic Programming*, 19–20, Supplement 1:629–682, 5 1994.

[Mug99] Stephen Muggleton. Inductive Logic Programming: Issues, results and the challenge of Learning Language in Logic. *Artificial Intelligence*, 114:283–296, 3 1999.