Metody sztucznej inteligencji

KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI

UMCS

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

Projekt „Programowa i strukturalna reforma systemu kształcenia na Wydziale Mat-Fiz-Inf”.
Projekt współfinansowany ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego.

Człowiek-najlepsza inwestycja

Uniwersytet Marii Curie-Skłodowskiej
Wydział Matematyki, Fizyki i Informatyki
Instytut Informatyki

Metody sztucznej inteligencji

Tomasz Żurek

Lublin 2011

Instytut Informatyki UMCS
Lublin 2011

Tomasz Żurek
Metody sztucznej inteligencji

Recenzent: Adrian Kapczyński

Opracowanie techniczne: Marcin Denkowski
Projekt okładki: Agnieszka Kuśmierska

Praca współfinansowana ze środków Unii Europejskiej w ramach
Europejskiego Funduszu Społecznego

Publikacja bezpłatna dostępna on-line na stronach
Instytutu Informatyki UMCS: informatyka.umcs.lublin.pl.

Wydawca
Uniwersytet Marii Curie-Skłodowskiej w Lublinie
Instytut Informatyki
pl. Marii Curie-Skłodowskiej 1, 20-031 Lublin
Redaktor serii: prof. dr hab. Paweł Mikołajczak
www: informatyka.umcs.lublin.pl
email: dyrii@hektor.umcs.lublin.pl

Druk
ESUS Agencja Reklamowo-Wydawnicza Tomasz Przybylak
ul. Ratajczaka 26/8
61-815 Poznań
www: www.esus.pl

ISBN: 978-83-62773-19-0

Spis treści

Przedmowa

Prawdopodobnie zawsze gdzieś w zakamarkach ludzkiej świadomości istniało, podszyte obawą i strachem, pragnienie stworzenia kogoś lub czegoś co miałoby w sobie jakąś namiastkę ludzkiego umysłu: praski Golem, Frankenstein, roboty R2-D2, C-3P0, czy rodzimy robot kolejkowy EWA 1. Pragnienie owo zawsze łączyło się z obawą. Obawą o utratę nad tym kontroli, ale także strachem przed poznaniem tego co w nas się dzieje i nieuprawnionym wkraczaniem w sferę sacrum. O ile w dawnych wiekach realizacja tego rodzaju marzeń była tylko sferą mitów i magii, o tyle dzisiaj, rozwój informatyki zdaje się stwarzać nadzieję, być może złudną, że coś takiego można rzeczywiście osiągnąć.

Sztuczna inteligencja jest działem informatyki, wokół którego narosło chyba najwięcej nieporozumień. Z jednej strony, dla laików, wygląda trochę jak niezrozumiała wiedza tajemna, która zaimplementowana daje zadziwiające rezultaty na pograniczu magii, z drugiej strony przez wiele osób postrzegana jest jako jedno wielkie szalbierstwo, które obiecując przysłowiowe gruszki na wierzbie zajmuje się wyłącznie sama sobą nie dając niczego co miałoby jakiekolwiek praktyczne zastosowanie. Jak jest naprawdę? Czym w rzeczywistości owa sztuczna inteligencja tak naprawdę jest?
Chcąc dyskutować o sztucznej inteligencji warto przede wszystkim wiedzieć czym jest inteligencja, i tu napotyka się pierwszy problem: co źródło to inna definicja. Pierwsza definicja pochodzi jakoby od Cycerona i tam definiuje się ją jako zdolności umysłowe człowieka (powtórzone za [14]), później różni autorzy różnie ją opisywali, m.in. jako zdolność do rozwiązywania problemów, zdolność do uczenia się, zdolność przystosowania się do okoliczności, dzięki dostrzeganiu abstrakcyjnych relacji, korzystaniu z uprzednich doświadczeń i skutecznej kontroli nad własnymi procesami poznawczymi, a także ogólna zdolność adaptacji do nowych warunków i wykonywania nowych zadań przez wykorzystanie środków myślenia. To tylko część definicji, które można znaleźć w internecie (m.in. pl.wikipedia.org), czy różnego rodzaju encyklopediach. Ponieważ samo pojęcie inteligencji jest mało precyzyjne tak też i zdefiniowanie sztucznej inteligencji nastręcza sporo trudności i każdy autor

definiuje ją sobie wg własnego uznania, przez co każdy patrzy z punktu widzenia jego własnych zainteresowań naukowych i inaczej ją widzi. Marvin Minsky jeden z twórców pojęcia sztucznej inteligencji określa ją jako naukę o maszynach realizujących zadania, które wymagają inteligencji, gdy są wykonywane przez człowieka (definicja podana za [14]), Edward Feigenbaum zwany ojcem systemów ekspertowych nazwał ją dziedziną informatyki dotyczącą metod i technik wnioskowania symbolicznego przez komputer oraz symbolicznej reprezentacji wiedzy stosowanej podczas takiego wnioskowania, w definicji tej skupił się na takim podejściu do sztucznej inteligencji jakie wykorzystane było w systemach ekspertowych (definicja za [14]). Z polskich autorów można przedstawić definicję prof. Włodzisława Ducha (z wykładów: [8]): „Sztuczna inteligencja to dziedzina nauki zajmująca się rozwiązywaniem zagadnień efektywnie niealgorytmizowalnych" i wielu wielu innych. Cechą wspólną tych definicji jest pewna „niedookreśloność" tzn. do końca nie wiadomo gdzie zaczyna się i gdzie kończy sztuczna inteligencja. Problem z nią jest taki, że wchodzi tam bardzo wiele bardzo różnych elementów, podejść dziedzin itp. Na wiele spraw tam poruszanych można patrzeć z bardzo różnych perspektyw. Gdyby te i inne definicje spróbować nieco uogólnić można by wyróżnić dwa podstawowe, kryjące się pod tym terminem, kierunki badań:

— Silna sztuczna inteligencja – która zajmuje się próbą realizacji „sztucznego mózgu", symulacji działania mózgu, myślenia ludzkiego, problemu świadomości itp. jest to dziedzina ciągle jeszcze w powijakach i daleko jest do tego, by udało się stworzyć choćby namiastkę „symulatora człowieka",
— Słaba sztuczna inteligencja – pod tym terminem zaś kryje się próba realizacji narzędzi, które wspomagają człowieka w zadaniach do których wykorzystuje swój intelekt. Ta dziedzina badań może się pochwalić znacznie większymi sukcesami i wieloma praktycznymi realizacjami.

Niniejsza praca koncentrować się będzie na zagadnieniach owej, słabej sztucznej inteligencji, wynika to m.in. ze względów praktycznych: właśnie zagadnienia tego kierunku badań znajdują znacznie większe zastosowanie w biznesie, medycynie, technice, prawie itp.

Podobnie jak z definicją sztucznej inteligencji, jest z próbą oceny kiedy komputer jest inteligentny. Do dziś uznaje się test Turinga (zainteresowanych na czym on polega odsyłam do internetu), ale ocenia on tylko pewien wycinek i to prawdopodobnie nie najważniejszy problemu oceny inteligencji...

Każdy ze wspomnianych wyżej kierunków badań oczywiście dzieli się na mniejsze „podkierunki". Do tego warto zwrócić uwagę na olbrzymią różnorodność narzędzi i technik kryjących się pod tymi terminami. Jest ich bardzo dużo i często się zdarza, że niektóre z nich po pewnym czasie zaczynają funkcjonować już jako osobna gałąź informatyki.

W związku z tym zamiast wgłębiać się w sens poszczególnych definicji lepiej będzie skupić się na tych cechach ludzkiej inteligencji, które warto byłoby w jakiś sposób zaimplementować w komputerze. Można wśród nich wyróżnić:
— Zdolność do uogólniania i analizy,
— Umiejętność znajdywania optymalnych rozwiązań dla nawet bardzo, dużych przestrzeni przeszukiwań,
— Umiejętność rozwiązywania i analizy konfliktów,
— Rozpoznawanie obrazów, dźwięków, mowy i generalnie obiektów w kontekstach,
— Zdolność do uczenia się,
— Umiejętność wnioskowania na bazie wiedzy ogólnej i specjalistycznej, rozumowanie,
— i wiele innych...

Do jakich, wobec tego, zadań wykorzystuje się sztuczną inteligencję?
— dokonywanie ekspertyz ekonomicznych (oceny inwestycji wniosków kredytowych itp.), prawnych, technicznych, medycznych,
— wspomaganie podejmowania decyzji ekonomicznych (doradzanie),
— rozpoznawanie obrazów, twarzy wzorców itp.,
— optymalizacja (harmonogramowanie, planowanie przewozów, alokacji zasobów finansowych itp.),
— generacja nowej wiedzy (poszukiwanie zależności między danymi, trendów, tendencji, reguł, drzew decyzyjnych itp.),
— prognozowanie zjawisk ekonomicznych, przyrodniczych itp.,
— rozumienie języka naturalnego,
— sterowanie urządzeniami,
— i wiele innych...

O czym, w związku z tym będzie ta książka? Z racji ograniczeń pojemnościowych będzie ona przedstawiała tylko pewien wybór tematów związanych ze sztuczną inteligencją. Zostaną tu przedstawione różne techniki i narzędzia wraz z przykładami zadań do których można je wykorzystać. Pozycja ta nie rości sobie także praw do bycia kompletną monografią opisywanych tu tematów. Z racji wspomnianych wyżej ograniczeń pewne zagadnienia przedstawione są tu dość wyczerpująco inne zaś stosunkowo pobieżnie, ale tak, by uświadomić czytelnikowi chociażby podstawy prezentowanej tematyki i dać mu szanse samodzielnego dokształcenia się. Narzędzia i techniki którymi posługuje się sztuczna inteligencja można w skrócie podzielić na dwie grupy:
— bazujące na, bądź wprost symulujące procesy biologiczne, jak: sieci neuronowe, algorytmy genetyczne itp.,
— bazujące na modelowaniu procesu rozumowania: generacja i reprezentacja wiedzy, wnioskowanie, przeszukiwanie i optymalizacja, logika itp.

Niniejsza praca koncentrować się będzie raczej na tej drugiej grupie, a takie

narzędzia jak sieci neuronowe, czy algorytmy genetyczne czytelnicy będą mogli poznać z innych, dostępnych pozycji.

W pierwszym rozdziale przedstawiono podstawy logiki jako narzędzia pozwalającego na opisanie procesów rozumowania i wyciągania wniosków. Treść rozdziału została wzbogacona przykładami opisu różnych sytuacji i procesów rozumowania.
Kolejny rozdział prezentuje język programowania w logice, czyli PROLOG. Przedstawione tam są podstawy języka i wbudowanego mechanizmu wnioskowania, zilustrowane przykładami problemów i ich rozwiązań. Język PROLOG jest językiem szczególnie predysponowanym do modelowania procesów reprezentacji wiedzy i wnioskowania, dlatego zaprezentowane w książce przykłady odnoszą się w głównej mierze do tej klasy problemów.
W trzecim rozdziale zaprezentowano jedno z najpopularniejszych narzędzi sztucznej inteligencji, czyli systemy ekspertowe. Przedstawione tam są sposoby reprezentacji wiedzy i mechanizmy wnioskowania. Duży nacisk został położony na ukazanie różnych sposobów wyrażania wiedzy niepewnej, niepełnej i sprzecznej. Przedstawione tez zostały formalne zasady i podstawy tworzenia ontologii pozwalających na reprezentację fragmentów wiedzy zdroworozsądkowej.
Kolejny rozdział dotyczy systemów uczących się, koncentrując się na jednym przykładowym algorytmie generacji drzew decyzyjnych z przykładów.
Rozdział piąty wraca do tematyki programowania opisując języki klasy *constraint logic programming*, czyli programowania w logice z ograniczeniami, będące rozwinięciem języka PROLOG. Na przykładzie języka ECLiPSe przedstawiono zasady działania wbudowanego mechanizmu wnioskowania, solwerów, oraz podstawowych predykatów ilustrując je przykładami typowych zadań przeszukiwania i optymalizacji, do których ów język jest szczególnie predysponowany.
Ostatni rozdział skupia się na przedstawieniu podstaw teorii gier, czyli dyscypliny zajmującej się modelowaniem i rozwiązywaniem sytuacji konfliktowych. Wykorzystanie narzędzi teorii gier jest szczególnie przydatne w modelowaniu sytuacji konfliktowych w biznesie i ma dużą przyszłość w praktycznych realizacjach systemów sztucznej inteligencji.

Autor ma nadzieję, że niniejsza pozycja będzie ciekawym uzupełnieniem wykładów z przedmiotu „Metody sztucznej inteligencji", pozwoli wejść w fascynujący świat systemów sztucznej inteligencji i przyda się czytelnikom nie tylko do zdania egzaminu, ale także w czasie ich zmagań z problemami zawodowymi.

Rozdział 1

Logika i wnioskowanie

Początki logiki jako dyscypliny naukowej wywodzą się ze starożytnej Grecji. Służyła ona do formalnego opisu dedukcji, czyli wyprowadzania prawdziwych stwierdzeń („wniosków") ze stwierdzeń uznawanych za prawdziwe („przesłanek") [2]. Od początku była traktowana jako dział filozofii, dopiero w XIX w. zaczęto patrzyć na nią jako także część matematyki. Najprościej można ją zdefiniować jako naukę o rozumowaniu [25], Wikipedia podaje, że logika (gr. logos - rozum) to nauka normatywna, analizująca źródła poznania pod względem prawomocności czynności poznawczych z nimi związanych. Zajmuje się badaniem ogólnych praw, według których przebiegają wszelkie poprawne rozumowania, w szczególności wnioskowania. Logika, jako dyscyplina normatywna, nie tylko opisuje jak faktycznie przebiegają rozumowania, ale także formułuje twierdzenia normatywne, mówiące o tym, jak rozumowania powinny przebiegać. Logika jako nauka opisująca zasady poprawnego rozumowania znalazła szczególne zastosowanie w sztucznej inteligencji, przede wszystkim zaś w narzędziach których zasada działania opierała się na reprezentacji wiedzy i rozumowaniu.
Zasady rozumowania jakimi posługuje się człowiek często wykraczają poza dobrze sformalizowane zasady logiki, w związku z czym podczas modelowania procesów rozumowania w systemach informatycznych zazwyczaj mamy do czynienia z pewnym uproszczeniem rzeczywistych procesów myślowych. Z drugiej strony rozwój systemów sztucznej inteligencji, sposobów reprezentacji wiedzy i wnioskowania wymusza także rozwój logiki, która pozwala na formalizację mechanizmów rozumowania.
Warto zwrócić uwagę na pewien aspekt związany z wykorzystaniem zasad logiki w budowie narzędzi sztucznej inteligencji: Termin „logika" i stwierdzenie, że coś jest logiczne w języku potocznym często jest mylnie utożsamiane z tym, że jest sensowne, spójne, zgodne z rzeczywistością, co nie do końca musi być prawdą: logika nie analizuje treści wypowiedzi i niejednokrotnie zdanie poprawne pod względem logicznym jest całkowicie pozbawione sensu w zdroworozsądkowym tego słowa znaczeniu. Logika bada jedynie poprawność i wartość logiczną zdań natomiast nie ma żadnych mechanizmów analizy i rozumienia treści owych zdań. W związku z tym należy pamiętać, że logika jest tylko pewnym narzędziem wspomagającym i ułatwiającym budowę narzędzi sztucznej inteligencji ale sama z siebie nie jest ani trochę „inteligentna" w potocznym znaczeniu tego słowa.

W poniższym rozdziale zostaną przedstawione podstawy logiki konieczne do zrozumienia zagadnień reprezentacji wiedzy i rozumowania. Niniejsze opracowanie nie rości sobie praw do bycia kompletnym wprowadzeniem do logiki, logiki matematycznej ani nawet logiki pierwszego rzędu. Stanowi jedynie pewien wstęp pozwalający na szersze spojrzenie na problem reprezentacji wiedzy i wnioskowania.

1.1. Rachunek zdań

Rachunek zdań jest działem logiki matematycznej, w którym bada się związki między zdaniami, które nie mają wewnętrznej struktury, lub funkcjami zadaniowymi (które są utworzone ze zdań lub innych funkcji zdaniowych oraz spójników). W klasycznym rachunku zdań każdemu zdaniu można przypisać jedną z dwóch wartości: prawdę lub fałsz. Treść zdań nie ma znaczenia, o prawdziwości formuł decyduje wyłącznie prawdziwość bądź fałszywość zdań składowych oraz funkcje prawdy użytych spójników [2]. W klasycznym rachunku zdań mamy do czynienia z:
— zmiennymi zdaniowymi (zdaniami),
— funktorami (spójnikami, operacjami),
— symbolami pomocniczymi (nawiasy).
Funktory mogą być jedno i wieloargumentowe, stałe logiczne: fałsz i prawda też można traktować jako szczególny bezargumentowy funktor [25], przy czym należy rozróżnić stałe logiczne od wartości logicznej jaką przyjmują zmienne zdaniowe, stała logiczna prawda ma zawsze wartość logiczną prawda. W niniejszej pracy rozpatrywane będą funktory jedno i dwuargumentowe. Ze względu na ograniczony zakres wartości jakie mogą przyjmować zdania istnieje skończony zbiór n- argumentowych funktorów. Funktorów jednozdaniowych jest dokładnie cztery: Warto zauważyć, że funktor f2 jest

Tabela 1.1. Funktory jednoargumentowe

X	f1	f2	f3	f4
0	1	1	0	0
1	1	0	1	0

identycznością, czyli odwzorowuje argument w samego siebie, natomiast funktory f1 i f4 są funkcjami stałymi. Tego rodzaju funktory można nazwać trywialnymi. Najciekawszym funktorem jednoargumentowym jest funktor f3, czyli negacja. Funktorów dwuargumentowych jest znacznie więcej bo aż 16: Część z powyższych funktorów podobnie jak w przypadku funktorów

Tabela 1.2. Funktory dwuargumentowe

X	X	f1	f2	f3	f4	f5	f6	f7	f8	f9	f10	f11	f12
1	1	1	1	1	1	1	1	1	1	0	0	0	0
1	0	1	1	1	1	0	0	0	0	1	1	1	1
0	1	1	1	0	0	1	1	0	0	1	1	0	0
0	0	1	0	1	0	1	0	1	0	1	0	1	0

jednoargumentowych ma charakter trywialny, jednakże w pozostałej części jest kilka szczególnie interesujących: Powyższe funktory są różnie nazywane,

Tabela 1.3. Funktory dwuargumentowe, c.d.

X	X	f13	f14	f15	f16
1	1	0	0	0	0
1	0	0	0	0	0
0	1	1	1	0	0
0	0	1	0	1	0

Tabela 1.4. Najważniejsze funktory dwuargumentowe

X	X	lub	i	implikacja	równoważność	xor
1	1	1	1	1	1	0
1	0	1	0	0	0	1
0	1	1	0	1	0	1
0	0	0	0	1	1	0

najpopularniejsze są jednak określenia z języka naturalnego: *lub* (alternatywa), *i* (koniunkcja), *implikuje*, *jest równoważne*, *albo* (odwrotna koniunkcja). Funktory te zostały tak nazwane przede wszystkim na ich bliskość znaczeniową względem tych pojęć z języka naturalnego. Niestey język naturalny jest znacznie mniej precyzyjny niż język logiki, przez co niejednokrotnie w potocznych sformułowaniach nie zawsze mają tożsame znaczenie.
Zmienne zdaniowe i funktory łączą się w formuły zdaniowe, które definiowane są następująco [25]:
— Zmienna zdaniowa jest formułą zdaniową,
— stałe: fałsz i prawda są formułą zdaniową,
— jeżeli p jest formułą zdaniową to $\neg p$ też jest formułą zdaniową,
— jeśli p i q są formułami zdaniowymi to wyrażenie p operator q, gdzie operator jest jednym z funktorów: $\vee$, $\rightarrow$, $\leftrightarrow$, $\wedge$, $\oplus$, jest też formułą zdaniową.

Za pomocą formuł można już modelować pewne wypowiedzi z języka naturalnego: Janek lubi Zosię $\wedge$ Marek lubi Marysię – co odczytuje się jako Janek lubi Zosię i Marek lubi Marysię.
Na obiad będzie zupa $\vee$ na obiad będą naleśniki – co odczytuje się jako: na obiad będzie zupa lub na obiad będą naleśniki, czyli może być zupa, mogą być naleśniki i mogą być oba dania.

Wyrażeniom w języku naturalnym często brakuje precyzji i słowa o podobnym ale nie takim samym znaczeniu bywają stosowane zamiennie, zdanie: jutro o godz. 9 pójdę na zajęcia lub jutro o 9 pójdę na wagary nie można przetłumaczyć na:

jutro o godz. 9 pójdę na zajęcia $\vee$ jutro o 9 pójdę na wagary,

ze względu na to, że nie jest możliwa sytuacja w której autor wypowiedzi będzie jednocześnie na zajęciach i na wagarach. Bardziej pasuje tu życie funktora odwrotnej koniunkcji czyli:

jutro o godz. 9 pójdę na zajęcia $\oplus$ jutro o 9 pójdę na wagary.

W formułach rachunku zdań aby uporządkować zapis zwykle wykorzystuje się również nawiasy, natomiast jeśli nawiasów nie ma to najwyższy priorytet ma negacja, potem na jednakowym poziomie koniunkcja, alternatywa i alternatywa wykluczająca, najniższy priorytet ma implikacja i równoważność.
Wartościowaniem zmiennej zdaniowej nazywa się funkcję przypisującą konkretnej funkcji konkretną wartość logiczną, natomiast funkcję przypisującą wartość logiczną formule nazywa się interpretacją [2]. Aby obliczyć wartość logiczną formuły zdaniowej konieczna jest znajomość wartościowania wszystkich zmiennych zdaniowych w niej występujących oraz funkcje prawdy poszczególnych funktorów. Jeżeli formuła ma postać:

$$(p \vee q) \wedge (\neg p \vee \neg q)$$

to dla wartościowania w(X), dla którego w(p) = 1 i w(q) = 0, jej interpretację i(X) można wyznaczyć z funkcji prawdy dla cząstkowych formuł:

$$i((p \vee q)) = 1$$
$$i(\neg p) = 0$$
$$i(\neg q) = 1$$
$$i((\neg p \vee \neg q)) = 1$$
$$i((p \vee q) \wedge (\neg p \vee \neg q)) = 1$$

Niektóre formuły rachunku zdań są tak skonstruowane, że niezależnie od wartościowania zmiennych zdaniowych zawsze są prawdziwe. Taki rodzaj formuł nazywa się tautologiami, bądź prawami rachunku zdań. Część tych praw ma duże znaczenie w reprezentacji wiedzy w systemach sztucznej inteligencji zapewniając spójność i niesprzeczność formuł oraz intuicyjną zgodność z tzw. zdrowym rozsądkiem. Kilka przykładów najważniejszych praw wraz z komentarzem zaprezentowanych jest poniżej:
— Prawo tożsamości dla implikacji: p $\rightarrow$ p; czyli z p wynika p,
— Prawo tożsamości dla równoważności: p $\leftrightarrow$ p; czyli p jest równoważne z p,

— Prawo podwójnego przeczenia: $p \leftrightarrow \neg(\neg p)$; czyli p jest równoważne z nie nie p,
— Prawo wyłączonego środka: $p \vee \neg p$; czyli prawdą jest p lub nie p. Jest to bardzo ważne prawo, które pozwala zachować niesprzeczność formuł,
— Prawo wyłączonej sprzeczności: $\neg(p \wedge \neg p)$; nieprawdą jest, że jest p i nie jest p,
— Prawo przemienności alternatywy: $(p \vee q) \leftrightarrow (q \vee p)$,
— Prawo przemienności koniunkcji: $(p \wedge q) \leftrightarrow (q \wedge p)$,
— Prawo łączności alternatywy: $[(p \vee q) \vee r] \leftrightarrow [p \vee (q \vee r)]$,
— Prawo łączności koniunkcji: $[(p \wedge q) \wedge r] \leftrightarrow [p \wedge (q \wedge r)]$,
— Prawa De Morgana: $\neg (p \vee q) \leftrightarrow (\neg p \wedge \neg q)$ oraz $\neg (p \wedge q) \leftrightarrow (\neg p \vee \neg q)$,
— Prawo odrywania: $[(p \rightarrow q) \wedge p] \rightarrow q$; jeśli z p wynika q i prawdą jest p, to prawdą jest q. Jest to bardzo ważna reguła, która pozwala na realizację mechanizmów wnioskowania.

Jeżeli pewne formuły rachunku zdań, niezależnie od wartościowania są sobie równoważne to takie formuły nazywa się logicznie równoważnymi. Kilka przykładów formuł logicznie równoważnych zostało również zaprezentowanych powyżej. Dobrym ćwiczeniem może być udowodnienie, że formuły te są równoważne oraz poszukanie innych formuł logicznie równoważnych. Przyglądając się powyższym tautologiom można łatwo zauważyć, że każdy funktor można zastępować odpowiednią kombinacją innych funktorów np.: $p \rightarrow q$ można zastąpić (jest równoważne) formułą $\neg p \vee q$. W ten sposób, w celu np. uproszczenia procesu wnioskowania, można zredukować liczbę funktorów. Możliwości wykorzystania rachunku zdań w reprezentacji są bardzo ograniczone, jedyna informacja jaką dla systemu niesie każde zdanie, jest jego wartościowanie, nie można zdefiniować tu żadnych właściwości, relacji między zdaniami ,w związku z czym system informatyczny nie ma zbyt szerokich możliwości wyrażania i przetwarzania wiedzy. Jedyne co można wywnioskować to interpretacje dla różnych funkcji zdaniowych, przez co możliwości wykorzystania klasycznego rachunku zdań są bardzo niewielki.

1.2. Logika pierwszego rzędu

Mając na względzie ograniczone możliwości klasycznego rachunku zdań warto przyjrzeć się systemowi logicznemu o nieco szerszych możliwościach reprezentacji wiedzy.

Rachunek predykatów pierwszego rzędu (bądź logika pierwszego rzędu) pomimo, iż podobnie jak rachunek zdań ma charakter dwuwartościowy (prawda i fałsz) i korzysta z tych samych funktorów (koniunkcja, alternatywa, implikacja, równoważność, itd.) wprowadza szereg nowych elementów: przede

wszystkim w systemie pojawiają się zmienne nazwowe, kwantyfikatory (ogólny i egzystencjalny), znak równości i predykaty.
Generalnie rachunek predykatów pierwszego rzędu można traktować jako rozszerzenie rachunku zdań, pozwala on zdefiniować właściwości obiektów, relacje między nimi i, za pomocą kwantyfikatorów odwołujących się do całej ich zbiorowości, modelować te relacje.
Jest to system logiczny, w którym kwantyfikatory mogą dotyczyć zmiennych których domenę stanowią zbiory obiektów, a nie zbiory zbiorów obiektów, bądź funkcje. W związku z tym w logice pierwszego rzędu można wyrazić zdanie:„Każdy obywatel ma numer PESEL" ale nie można wyrazić zdania „każdy reguła, który dotyczy obywatela może być złamana" dlatego, że w tym drugim zdaniu kwantyfikator dotyczy reguły, czyli funkcji, której argumentem jest obywatel.
W logice pierwszego rzędu korzysta się z:

— zmiennych – za które będą podstawiane nazwy obiektów,
— stałych – nazwy obiektów,
— predykatów – które reprezentują właściwości obiektów i relacje między obiektami,
— funktorów – koniunkcji, alternatywy, implikacji, równoważności, zaprzeczenia i równości,
— kwantyfikatorów – kwantyfikatora ogólnego (∀x): „dla każdego x" i szczegółowego (∃x): „istnieje x", gdzie x jest zmienną,
— znaków pomocniczych (nawiasy).

Język logiki pierwszego rzędu:
Niech P będzie przeliczalnym zbiorem predykatów. Predykaty mogą być bezargumentowe, bądź mieć dowolną skończoną ilość argumentów.
Niech A będzie przeliczalnym zbiorem stałych.
Niech V będzie przeliczalnym zbiorem zmiennych.

— dowolna zmienna x ze zbioru V jest termem,
— dowolna stała a ze zbioru A jest termem,
— Jeżeli t1..tn są termami a p jest predykatem n-argumentowym ze zbioru P to p(t1,...,tn) jest atomem lub formułą atomową,
— Jeżeli p jest bezargumentowym predykatem ze zbioru P to p jest atomem,
— Atom jest formułą,
— Wyrażenie: „nie formuła" (¬formuła) jest formułą,
— Wyrażenie: „formuła funktor formuła" jest formułą,
— Wyrażenie: „∀x formuła" jest formuła (x jest zmienną ze zbioru V),
— Wyrażenie: „∃x formuła" jest formułą (x jest zmienną ze zbioru V).

Zdanie w języku pierwszego rzędu to taka formuła, w której każda zmienna

jest związana, czyli znajduje się w zasięgu działania jakiegoś kwantyfikatora. Jeśli jakaś zmienna w formule nie jest związana to nazywa się ją zmienną wolną a wystąpienia jej wystąpieniami wolnymi.

1.2.1. Semantyka

Logika pierwszego rzędu pozwala na wprowadzenie pewnej namiastki semantyki do wyrażeń języka poprzez formułowanie predykatów, które mogą odnosić się do cech i relacji świata rzeczywistego. Obiekty i predykaty same z siebie nie wyrażają żadnej treści, można jednak powiązać je z pewnymi konkretnymi obiektami, właściwościami i relacjami ze świata rzeczywistego i wykorzystać je do tworzenia modelu sytuacji ze świata rzeczywistego w następujący sposób:

— W świecie istnieją obiekty.
— Każdy jednoargumentowy predykat może być spełniony (prawdziwy) dla części obiektów, dla części zaś nie. Jeśli dany predykat jest spełniony dla danego obiektu to mówi się, że dany obiekt ma pewną (opisywaną przez ten predykat) właściwość. Obiekty, dla których predykat ów nie jest prawdziwy nie mają tej właściwości.
— Dla predykatów wieloagumentowych mówi się o relacji między obiektami, którą opisuje dany predykat i jeśli dla pewnego n-argumentowego predykatu, dla grupy n-obiektów w konkretnym układzie ten predykat jest spełniony, to dana relacja między tymi obiektami zachodzi (jest prawdziwa), w przeciwnym razie nie.

Bazując na tych trzech predykatach (*dziewczyna, chłopak, lubi*), dwóch stałych (*Jaś, Zosia*) oraz opisanej wyżej metodyki przypisania semantyki predykatom i stałym można wyrazić pewną treść: Stała *Zosia* reprezentuje obiekt świata rzeczywistego Zosia, stała *Jaś* reprezentuje rzeczywisty obiekt Jaś, predykat *dziewczyna* oznacza, że obiekt, który jest argumentem jest dziewczyną (ma właściwość bycia dziewczyną), predykat *chłopak* oznacza, że obiekt będący jego argumentem jest chłopakiem (ma właściwość bycia chłopakiem) a dwuargumentowy predykat *lubi* oznacza, że zachodzi relacja lubienia między argumentem pierwszym a drugim. Zosia jest dziewczyną, Jaś chłopakiem oraz Jaś lubi Zosię.

dziewczyna(Zosia).
chłopak(Jaś).
lubi(Jaś, Zosia).

Można z powyższego zapisu odczytać pewne fakty nie zdefiniowane wprost: Jaś nie spełnia właściwości bycia dziewczyną, Zosia nie spełnia właściwości bycia chłopakiem, nie zachodzi relacja „lubi" między Zosią i Jasiem (z tego co zostało zadeklarowane wynika, że Jaś lubi Zosię, nigdzie nie jest

zadeklarowane, że Zosia odwzajemnia te uczucia). Z powyższych założeń wynika jeszcze jedna, nie zdefiniowane wprost założenie (choć wynikające z zasad przypisywania znaczenia predykatom): jeśli nie jest zadeklarowane, że dla danych argumentów dany predykat jest spełniony traktuje się go jako nieprawdę – okaże się ono, w nieco ogólniejszej wersji, bardzo ważne dla wszystkich systemów bazujących na wiedzy. Oczywiście semantyka odnosi do tego jak człowiek może zrozumieć wynik wnioskowania i nie powoduje, że z zasad logiki możliwe będzie wyprowadzenie automatycznie sensu jakiegokolwiek wyrażenia. Użycie stałej *Zosia* do oznaczenia konkretnej koleżanki o takim imieniu nie spowoduje, że w jakikolwiek sposób można odnieść się do tej rzeczywistej osoby. Użycie predykatu *dziewczyna(Zosia)* może przypisać stałej *Zosia* cechę o nazwie dziewczyna. Cechę tą możemy oczywiście zinterpretować tak, że uznamy ową Zosię za dziewczynę, ale znaczenie jakie nadajemy tej cesze jest poza zainteresowaniem logiki: cechę tą można by nazwać np.: zGwsr3t i nie zmieniło by to, z punktu widzenia logiki, sensu wyrażenia.

Aby precyzyjniej wyjaśnić pojęcia używane w logice pierwszego rzędu konieczne będzie przedstawienie kilku definicji:

Interpretacja Niech F będzie zbiorem formuł, zbiór P jest zbiorem wszystkich predykatów występujących w formułach zbioru F, zaś A będzie zbiorem stałych występujących w F. Trójkę:

$$(D, \{R_1 \dots R_m\}, \{d_1 \dots d_m\})$$

gdzie D jest niepustą domeną, , Ri jest n-argumentową relacją określoną na D, przyporządkowaną predykatowi p_i ze zbioru P, a d_i są elementami domeny przyporządkowanymi stałym ze zbioru A[2].

Wartościowanie zmiennej

Wartościowaniem zmiennej nazywa się funkcję określoną na zbiorze zmiennych V przypisującą każdej zmiennej v_i ze zbioru V wartość z domeny D.

Wartość formuły

Dla określonej interpretacji I i wartościowania W zmiennych wartość formuły F oznaczoną jako w(F) określa się w następujący sposób:

— Jeśli F jest formułą atomową, $p(c_1...c_n)$ jest predykatem ze zbioru predykatów P, każde c_i jest zmienną x_i bądź stałą a_i, wartość w(F) = 1 wtedy i tylko wtedy, gdy $(d_1...d_n)$należące do R_k, gdzie R_k jest relacją przyporządkowania w interpretacji I predykatowi p, a d_i są elementami dziedziny przyporządkowanymi c_i przez interpretację (gdy c_i jest stałą) albo wartościowanie (gdy c_i jest zmienną),

— w(¬A) = 1 wtedy i tylko wtedy, gdy w(A) = 0,

— w(A1 ∨ A2) =1 wtedy, gdy w(A1) = 1 lub w(A2) = 1. Analogicznie dla innych operatorów logicznych,
— w(∀xA) =1 wtedy i tylko wtedy, gdy w(A) = 1 dla każdego d należącego do D,
— w(∃xA) =1 wtedy i tylko wtedy, gdy w(A) = 1 dla pewnego d należącego do D [2].

Nieco mniej formalnie można stwierdzić, że wartość formuły jest zmienną logiczną przypisaną tej formule dla określonej interpretacji i wartościowania.

Podobnie jak w rachunku zdań, w logice pierwszego rzędu można zdefiniować grupę formuł zawsze prawdziwych, czyli tautologii. Poniżej zaprezentowano przykłady takich formuł (x, y to zmienne A, B to predykaty):

$$\forall x(A \to B) \to (\exists xA \to \exists xB)$$

$$\neg\forall xA \leftrightarrow \exists x\neg A$$

$$\neg\exists xA \leftrightarrow \exists x\neg A$$

$$\forall x(A \vee B) \leftrightarrow \forall xA \wedge \forall xB$$

$$\exists x(A \vee B) \leftrightarrow \exists xA \vee \exists xB$$

$$\forall xA \to \exists xA$$

$$\forall x\forall yA \leftrightarrow \forall y\forall xA$$

$$\exists x\exists yA \leftrightarrow \exists y\exists xA$$

$$\exists x\forall yA \to \forall y\exists xA$$

Poza wspomnianymi wyżej tautologiami w logice pierwszego rzędu funkcjonują również tautologie znane z rachunku zdań, w tym szczególnie istotne dla modelowania rozumowania zasady wyłączonego środka i prawo odrywania.

Nieuważne wykorzystanie logiki pierwszego rzędu do modelowania wyrażeń z języka potocznego (Podobnie jak w rachunku zdań) może prowadzić do błędów. Funktory i kwantyfikatory są precyzyjnie zdefiniowane i przyjmuje się je jako odpowiedniki spójników i kwantyfikatorów z języka naturalnego i właśnie owo precyzyjne zdefiniowanie często przeszkadza w bezpośrednim przekładzie wyrażeń potocznych, w których bardzo często aż roi się od niekonsekwencji i niejednoznaczności. W związku z tym podczas modelowania sytuacji rzeczywistych za pomocą logiki pierwszego rzędu należy położyć bardzo duży nacisk na to aby jak najdokładniej przełożyć treść wyrażenia z języka naturalnego na język logiki. Bardzo częstym błędem jest utożsamianie implikacji logicznej z regułą typu „jeżeli ... to ...", w której jest wyraźnie oznaczona przyczyna (jeżeli...) i skutek (to ...). W implikacji nie

analizuje się treści formuła atomowych a jedynie ich prawdziwość, w związku z czym można powiedzieć, że: „jeżeli pada deszcz to jest niskie ciśnienie", co można wyrazić tak: *pada deszcz* $\rightarrow$ *niskie ciśnienie* ale równocześnie można powiedzieć, że „jeżeli jest niskie ciśnienie to pada deszcz", co można wyrazić tak: *niskie ciśnienie* $\rightarrow$ *pada deszcz* oba te wyrażenia są prawidłową implikacją, ale w żaden sposób powyższe wyrażenia nie wyjaśniają co co jest tak naprawdę przyczyną czego i równie dobrze można by zadeklarować, że: *ziemia jest okrągła* $\rightarrow$ *studenci lubią chodzić na wagary* Czyli: „jeżeli Ziemia jest okrągła to studenci lubią chodzić na wagary". Takie wyrażenie będzie prawidłową implikacją ale nie jest prawidłową regułą, dlatego że nie ma związku przyczynowo skutkowego między tym, że Ziemia jest okrągła, a upodobaniem studentów do lenistwa.
Inne rodzaje błędów mogą powstać w sytuacji niejednoznaczności języka naturalnego: „każdy, kto gra w karty i pije alkohol będzie miał zszarganą reputację", co można wyrazić tak: $\forall x(gra\ w\ karty(x) \wedge pije\ alkohol(x) \rightarrow zszargana\ reputacja(x))$ Z powyższego wyrażenia wynika, że jeśli ktoś grał w karty ale nie pił alkoholu to nie wiadomo, czy będzie miał zszarganą reputację. Zdrowy rozsądek podpowiada więc, że poprawniej będzie jeśli w zdaniu użyty będzie spójnik „lub". Inny rodzaj nieprawidłowości przy przekładzie zwrotów z języka naturalnego na język logiki pierwszego rzędu może wynikać z mylenia koniunkcji z implikacją w zasięgu kwantyfikatora [25]. Dwa bardzo podobne zdania, np.: „każdy student ma indeks", „pewien student ma indeks" wyrażone w logice pierwszego rzędu różnią się dość mocno: $\forall x(student(x) \rightarrow ma\ indeks(x))$ $\exists x(student(x) \wedge ma\ indeks(x))$ Wykorzystanie w drugim wyrażeniu koniunkcji zamiast implikacji lepiej oddaje specyfikę tego zdania. Osoby dociekliwe mogą znaleźć jeszcze wiele innych przykładów, w których bezpośrednie przełożenie wyrażenia z języka naturalnego na język logiki pierwszego rzędu nie oddaje jego właściwej treści ciekawe przykłady można znaleźć np. tu [25]

1.2.2. Reguły wnioskowania

Regułą wnioskowania nazywa się przekształcenie zbioru formuł P, w zbiór formuł W, takie, że jeżeli formuły P są prawdziwe w danej interpretacji, to wszystkie formuły W też są prawdziwe w tej interpretacji. Formuły ze zbioru P nazywa się przesłankami natomiast formuły ze zbioru W określa się jako wnioski.
Regułę wnioskowania zapisuje się tak:

$$\frac{P}{W}$$

Najprostszą regułą wnioskowania jest reguła odrywania (modus ponens) ma ona taką postać:

$$\frac{P \to W, P}{W}$$

czyli przykładowo:

$$\frac{\text{świeci słonce} \to \text{idę na wagary},\ \text{świeci słońce}}{\text{idę na wagary}}$$

co można wyrazić w języku naturalnym: jeżeli świeci słońce to pójdę na wagary, ponieważ prawdą jest, że świeci słońce, to prawdą jest, że pójdę na wagary. W wersji ze zmienną reguła odrywania wygląda tak:

$$\frac{P(x) \to W(x), P(a)}{W(a)}$$

czyli przykładowo:

$$\frac{\text{student(x)} \to \text{lubi codzić na wagary(x)},\ \text{student(Zenek)}}{\text{lubi chodzić na wagary(Zenek)}}$$

co można przełożyć na język naturalny jako:
Każdy student lubi chodzić na wagary, Zenek jest studentem, wobec czego Zenek lubi chodzić na wagary.
Poniżej zaprezentowano kilka definicji pomocnych w zrozumieniu dalszej części niniejszej pozycji:

Aksjomat
Formuły z założenia prawdziwe w pewnej teorii T noszą nazwę aksjomatów.

Wnioskowanie
Jeżeli dla pewnego zbioru aksjomatów, formuł oraz reguł wnioskowania możliwe jest zbudowanie ciągu formuł takich, że poprzez zastosowanie reguły wnioskowania do aksjomatów lub formuł jako przesłanek otrzyma się inne formuły jako wnioski, to taki ciąg formuł nazywa się wnioskowaniem.

Literał
Formułę atomową lub formułę atomową z negacją nazywa się literałem (jeśli formuła w literale jest z negacją mówi się o literale negatywnym, w przeciwnym razie o literale pozytywnym).

Klauzula
Klauzulą nazywa formułę, przyjmującą postać:

$$(\forall x_1)...(\forall x_m)(L_1 \vee \ldots \vee L_n)$$

Klauzula pusta, jest zawsze fałszywa. Jeśli w klauzuli nie użyto innych zmiennych poza x_1 .. xm to zazwyczaj kwantyfikatory się pomija.

Klauzula Horna to klauzula, która zawiera przynajmniej jeden pozytywny literał. Klauzulę Horna można przedstawić jako:

$$\neg L_1 \vee \ldots \vee \neg L_{n-1} \vee L_n$$

lub:

$$L_1 \wedge \ldots \wedge L_{n-1} => L_n$$

Przypisanie
Przypisanie jest to para *(x, t)*, gdzie x to zmienna, a t to term, który stanowi wartość zmiennej x.
Podstawienie
Podstawieniem zaś nazywa się zbiór przypisań termów do wszystkich zmiennych. Wynikiem zastosowania podstawienia do formuły jest zastąpienie wszystkich zmiennych przypisanymi im termami.

Logika pierwszego rzędu jest narzędziem o dość dużej sile wyrazu, na tyle dużej, że można ją wykorzystywać do modelowania wielu sytuacji z rzeczywistości. Szczególnie istotna jest możliwość wykorzystania kwantyfikatorów, które pozwalają na definiowanie zależności o ogólnym charakterze, oraz predykatów, które pozwalają na zdefiniowanie cech i relacji między obiektami. Wykorzystanie tych narzędzi pozwala na opisanie różnych nietrywialnych zależności np. chcąc przypisać jakąś cechę (np. kolor) konkretnemu obiektowi (np. samochodowi ford) można napisać:

niebieski(ford).

Relacje między obiektami, np.: „Jan posiada (relacja własności) samochód ford" można wyrazić w taki sposób:

posiada(Jan, ford).

Istnienie relacji posiada(Jan, ford) w żadnym stopniu nie implikuje istnienia relacji

posiada(ford, Jan).

Czyli to, że Jan posiada samochód ford nie oznacza, że samochód ford posiada Jana. Jest to intuicyjnie zrozumiałe i dla takiej relacji zgodne ze zdrowym rozsądkiem, ale dla innych może już nie do końca oddawać stan faktyczny, np.:

krewny(Jan, Jurek).

Nie jest tożsamy z:

krewny(Jurek, Jan).

Choć zdroworozsądkowo oczywiście Jurek powinien być krewnym Jana, skoro Jan jest krewnym Jurka. Za pomocą predykatu można wyrazić także powiązania zachodzące między więcej niż dwoma obiektami, np. to, że Jan wręczył Zosi kwiaty:

$$\text{wręczył(Jan, Zosia, kwiaty).}$$

To jak taki predykat należy rozumieć zależy tylko od przyjętej konwencji. W powyższym przykładzie pierwszy argument opisywał kto, drugi komu a trzeci co było wręczone. Nic oczywiście nie stoi na przeszkodzie temu aby przyjąć inny układ np.: pierwszy argument opisuje co, drugi komu a trzeci kto wręczy dany przedmiot:

$$\text{wręczył(kwiaty, Zosia, Jan).}$$

Ważne przy tym jest jedynie trzymanie się jednej konwencji, dzięki czemu możliwe będzie właściwe zrozumienie wyrażenia.
Powyższe przykłady opisują tylko konkretne sytuacje dotyczące konkretnych obiektów. Wykorzystanie zmiennych, kwantyfikatorów i implikacji pozwala na definiowanie bardziej ogólnych stwierdzeń, np.:

$$\forall x(\text{wręcza}(x, \text{Zosia}, \text{kwiaty}) \rightarrow \text{lubi(Zosia,}x)).$$

Co można rozumieć jako zdanie: Zosia lubi każdego, kto wręcza jej kwiaty. Podczas definiowania tego rodzaju zależności ważne jest aby dokładnie i precyzyjnie opisać zależność. Gdyby przełożyć na język logiki zdanie: „Jan daje kwiaty każdemu kogo lubi" w najprostszym wydaniu model wyglądał by tak:

$$\forall x(\text{lubi(Jan,}x) \rightarrow \text{wręcza(Jan, } x, \text{kwiaty})).$$

Wiedząc, że Jan lubi Zosię, czyli:

$$\text{lubi(Jan, Zosia).}$$

Możemy przypisać zmiennej x term Zosia i wywnioskować, że Jan wręczy Zosi kwiaty. Gdyby jednak wiadomo było, że Jan lubi nie tylko Zosię ale także piwo, czyli dodatkowo:

$$\text{lubi(Jan, piwo).}$$

To poprzez przypisanie zmiennej x termu piwo można się dowiedzieć, że Jan wręczy kwiaty piwu. Działanie takie (choć oczywiście jest możliwe) jest z punktu widzenia zdrowego rozsądku absolutnie bezsensowne. Błąd w powyższym wyrażeniu leżał w tym, że milcząco założono, a nie zdefiniowanym wprost, że odbiorca kwiatów powinien być człowiekiem, zresztą w zdaniu z

języka naturalnego użyty jest zaimek „kogo" co wyraźnie sugeruje, że odbiorcą powinien być człowiek. W związku z tym zdanie: „Jan daje kwiaty każdemu kogo lubi" powinno zostać zamodelowane tak:

$$\forall x(\text{lubi(Jan,}x) \wedge \text{człowiek}(x) \rightarrow \text{wręcza(Jan,}x\text{, kwiaty})).$$

W jednej formule można wiązać różnymi kwantyfikatorami kilka zmiennych, co pozwala na wyrażenie nieco bardziej skomplikowanych zależności. Dla przykładu zamodelowanie takiego powiedzenia: „na każdy towar znajdzie się kupiec" możliwe jest dzięki wykorzystaniu dwóch kwantyfikatorów i dwóch zmiennych:

$$\forall x \exists y(\text{towar}(x) \wedge \text{kupi}(y, x)).$$

Te same zależności można wyrażać za pomocą różnie skonstruowanych wyrażeń, np. wspomniany wcześniej przykład samochodu o kolorze niebieskim można wyrazić tak:

$$\text{niebieski(ford)}.$$

Bądź tak:

$$\text{kolor(ford, niebieski)}.$$

W tym drugim sposobie wyrażenia konkretnej właściwości obiektu ford wykorzystano predykat kolor(..), który pozwolił na nazwanie cechy (kolor), a jej wartość jest wyrażona poprzez drugi argument predykatu. To samo wyrażenie można jeszcze bardziej uogólnić:

$$\text{cecha(ford, kolor, niebieski)}.$$

Predykat cecha(...) ma trzy argumenty, z których pierwszy opisuje obiekt, który ma cechę, drugi nazwę cechy, trzeci zaś wartość tej cechy. Trzy powyższe sposoby reprezentacji tej samej sytuacji różnią się poziomem skomplikowania i uniwersalnością, gdyż logika pierwszego rzędu pozwala tylko na używanie kwantyfikatorów tylko względem zmiennych, w związku z czym jeśli nazwa cechy będzie argumentem a nie nazwą predykatu to możliwe objęcie jej zakresem kwantyfikatora. Dla pierwszego przykładu kwantyfikator mógłby obejmować tylko obiekt, który ma kolor niebieski („dla każdego, który ma kolor niebieski ..."), w drugim kwantyfikator mógłby obejmować obiekt lub kolor („dla każdego, który ma kolor niebieski ..." ale także „dla każdego koloru..."), trzeci sposób dodaje jeszcze możliwość kwantyfikowania po rodzaju cechy („dla każdej cechy ..."). Nie trzeba dodawać, że ten sposób wyrażania cechy jest najbardziej uniwersalny, cechuje się największą ekspresją i daje największą swobodę w reprezentacji sytuacji ze świata rzeczywistego, jednakże wybór odpowiedniej metody wyrażenia konkretnej treści powinien być uzależniony od konkretnych wymagań, tak aby nie komplikować wyrażeń ponad miarę.

Logika pierwszego rzędu pozwala na prostą reprezentację szeregu zdroworozsądkowych zależności między różnego rodzaju obiektami i relacjami, np.:

— Rozłączność – często dwie cechy bądź relacje są wzajemnie wykluczające, tzn. zajście jednej wyklucza zajście drugiej:

$$\forall x(\text{mężczyzna}(x) \rightarrow \neg\text{kobieta}(x)).$$

Jeśli x jest mężczyzną to nie jest kobietą

— Podtypy – cecha bądź relacja może być podklasą pewnej ogólniejszej cechy bądź relacji:

$$\forall x(\text{chirurg}(x) \rightarrow \text{lekarz}(x)).$$

Jeśli x jest chirurgiem to jest lekarzem

— Komplementarność – pewne cechy bądź relacje mogą być komplementarne względem siebie:

$$\forall x(\text{człowiek}(x) \rightarrow \text{kobieta}(x) \vee \text{mężczyna}(x)).$$

Jeśli x jest człowiekiem to jest kobietą lub mężczyzną

— Symetria – relacje mogą być symetryczne, czyli skoro zachodzą dla pary x, y to zachodzą dla pary y, x:

$$\forall xy(\text{małżeństwo}(x, y) \rightarrow \text{małżeństwo}(y, x)).$$

Jeśli x jest małżonkiem y to y jest małżonkiem x

— Inwersja – Relacje są przeciwne względem siebie:

$$\forall xy(\text{dziecko}(x, y) \rightarrow \text{rodzic}(y, x)).$$

Jeśli x jest dzieckiem y to y jest rodzicem x.

— Ograniczenie swobody – niektóre cechy i relacje odnoszą się tylko do obiektów mających określone cechy, lub będących w określonych relacjach:

$$\forall xy(\text{małżeństwo}(x, y) \rightarrow \text{człowiek}(y) \wedge \text{człowiek}(x)).$$

Jeśli x i y są małżeństwem to x jest człowiekiem i y jest człowiekiem

1.2.3. Wnioskowanie

W jednym z wcześniejszych punktów zdefiniowano reguły wnioskowania i sam proces wnioskowania. Jako przykład reguły wnioskowania podano regułę odrywania czyli modus ponens. Reguła odrywania mówi, że jeśli uznaje

się prawdziwość poprzednika implikacji to musimy uznać prawdziwość jej następnika, czyli przykładowo:

$$\forall x(\text{chirurg}(x) \rightarrow \text{lekarz}(x)).$$

Jan jest chirurgiem, wobec czego z reguły odrywania wynika, że Jan jest lekarzem.
Inną regułą wnioskowania jest reguła jest reguła modus tollens, która mówi, że jeśli b wynika z a i b jest fałszywe, to a również jest fałszywe, czyli:

$$\forall x(\text{chirurg}(x) \rightarrow \text{lekarz}(x)).$$

Jan nie jest lekarzem, wobec czego nie jest również chirurgiem.
Rezolucja jest metodą automatycznego dowodzenia twierdzeń opartą na przekształcaniu klauzul aż doprowadzi się do sprzeczności. W ten sposób można udowodnić, że dane twierdzenie nie jest spełnialne, lub też, co jest równoważne, że jego zaprzeczenie jest tautologią. Reguła odrywania jest szczególnym przypadkiem rezolucji.
Człowiek podczas rozumowania posługuje się powyższymi regułami wnioskowania ale nie tylko nimi. Wykorzystuje również całą gamę znacznie bardziej skomplikowanych i niejednoznacznych reguł – reguł, które nie zawsze prowadzą do poprawnych odpowiedzi a są znacznie trudniejsze do spójnego zamodelowania. Takie reguły to np.:

— Indukcja – wysnuwanie ogólnych twierdzeń ze szczególnych przypadków (skoro Jaś, Małgosia, i Zosia są studentami i każdy z nich uczy się 3 godziny dziennie to każdy student uczy się 3 godziny dziennie),
— Abdukcja – wysnuwanie wniosku na temat prawdziwości poprzednika implikacji z prawdziwości jej następnika (jeżeli student się uczył to dostał piątkę, ponieważ Jan dostał piątkę to znaczy, że się uczył.),
— Wnioskowania wynikające wartości domyślnych, gdy brak jest wiedzy na temat prawdziwości przesłanek (nic nie wiadomo na temat ptaka o imieniu Twitty ale zakłada się domyślnie, że ptaki latają, w związku z czym Twitty lata),
— Analogia – wnioskowanie na podstawie podobieństwa do poprzednich przypadków,
— Z większego na mniejsze – wnioskowanie: jeżeli A to tym bardziej B. Występuje ono w kilku odmianach m.in. skoro zabronione jest coś mniejszego to tym bardziej zabronione jest coś większego (jeżeli zabronione korzystanie ze ściąg na egzaminie to tym bardziej zabronione jest odpisywanie pracy od kolegi),
— Wnioskowanie instrumentalne – skoro prawda jest A to B, które jest konieczne do tego aby A było prawdą również musi być prawdą.

Wszystkie te metody wnioskowania są zawodne, czyli nie zawsze otrzymuje się z nich prawdziwe wnioski, poza tym w większości mają charakter podważalny (mogą zostać podważone przez inne reguły wnioskowania) i często niemonotoniczny tzn. nowe fakty mogą doprowadzić do powstania niespójności (naruszona zostanie zasada wyłączonego środka), bądź doprowadzają do uznania starych faktów za nieprawdę.
Metody te są niezwykle ciekawe i stanowią o sile i elastyczności wnioskowania ludzkiego ale ich zamodelowanie nastręcza wielu trudności. Zostaną one dokładniej omówione w dalszej części książki.

Rozdział 2

Programowanie w logice

Logika jako nauka o poprawnym rozumowaniu daje pewien (mocno uproszczony) mechanizm formalizacji procesów myślowych. Formalizację tych procesów można traktować jako pierwszy krok podczas realizacji narzędzi sztucznej inteligencji. Kolejnym krokiem do realizacji takich narzędzi powinna być realizacja mechanizmu implementacji wyrażeń logiki i procesów wnioskowania w komputerze. Bazując na tym, na bazie języka logiki pierwszego rzędu opracowano całkiem nowy paradygmat programowania.

Programowanie w logice opiera się na właśnie na rachunku predykatów pierwszego rzędu. Program ma postać klauzul logiki pierwszego rzędu. O programowanie w logice mówi się, że jest rodzajem programowania deklaratywnego tzn. zadaniem programisty nie jest wprowadzeniem do komputera sposobu rozwiązania problemu (algorytmu), jak jest w proceduralnych językach programowania, a maksymalnie precyzyjne opisanie zadania i problemu, który powinien zostać rozwiązany. Samym rozwiązaniem problemu powinien zająć się komputer wykorzystując do tego zasady logiki i wnioskowania. W programowaniu w logice kolejność klauzul i warunków w klauzulach nie powinna mieć wpływu na wnioski, w praktyce jednak okazuje się, że czasami kolejność odgrywa ona pewną rolę.

Najpopularniejszym językiem programowania będącym próbą praktycznej realizacji programowania w logice jest język PROLOG (PROgramming in LOGic). Język PROLOG powstał na początku lat 70 w ośrodkach badawczych Edynburgu i Marsylii. Główną ideą jaka przyświecała twórcom tego języka było stworzenie narzędzia ułatwiającego automatyczne dowodzenie twierdzeń i przetwarzanie języka naturalnego. Język ten mimo tego, że nie w pełni implementuje zasady programowania w logice to jest powszechnie jako taki traktowany (o tym dlaczego PROLOG nie w pełni jest językiem programowania w logice będzie później, więcej można też przeczytać np. tu [7]). Język ten bardzo dobrze się nadaje do modelowania zależności logicznych w komputerze, umożliwiając bardzo zwięzłą i czytelną reprezentację modelu logicznego problemu, zwalniając jednocześnie programistę z konieczności opracowywania algorytmu rozwiązania problemu.

Poniżej zostaną zaprezentowane podstawy języka PROLOG. Nie jest to oczywiście pełny kurs języka, który można znaleźć np. tu [7], bądź tu [1], zamiarem autora nie była też nauka języka (która i tak najlepiej wychodzi podczas samodzielnych ćwiczeń), ani nawet demonstracja specyficznego paradygmatu programowania. Celem autora było tu raczej pokazanie jak można modelować różnego rodzaju wiedzę, procesy wnioskowania, czy problemy logiczne: to wszystko co może być przydatne do symulowania procesów rozumowania wykonywanych przez człowieka. W praktyce bowiem okazuje się, że bardzo wiele rzeczywistych sytuacji decyzyjnych daje się zaimplementować (czasem w pewnym uproszczeniu) za pomocą logiki pierwszego rzędu, programowania w logice i wreszcie języka PROLOG.

2.1. Podstawy

Program w PROLOG-u jest zbiorem klauzul Horna:

$$L_1 \wedge \ldots \wedge L_{n-1} \rightarrow L_n.$$

Gdzie L_1 do L_{n-1} to poprzedniki klauzuli natomiast L_n to następnik klauzuli, klauzule mogą być z poprzednikami lub bez. Klauzule z poprzednikami reprezentują zależności, klauzule bez poprzedników reprezentują fakty (prawdą jest, że L). Gdy w klauzuli wstępują zmienne X_1 ... X_m to klauzulę interpretuje się tak:

$$\forall x_1 \ldots x_m (L_1 \wedge \ldots \wedge L_{n-1} \rightarrow L_n).$$

Klauzule bez następnika nie występują bezpośrednio w programie, natomiast można je wykorzystywać w konwersacji (po kompilacji programu) jako cel wnioskowania. Gdy w takiej klauzuli występują zmienne, interpretuje się ją tak:

$$\exists x_1 \ldots x_m (L_1 \wedge \ldots \wedge L_n).$$

Proces wnioskowania w języku Prolog opiera się na specyficznym rodzaju rezolucji:
Podstawowa reguła jest następująca: wiedząc że z P wynika Q oraz z R wynika S, wnioskujemy że z P wynika S, tylko wtedy, gdy Q i R dadzą się zunifikować.
Jeśli w wyrażeniach występują zmienne to w trakcie wnioskowania znajduje się takie wartości zmiennych dla których Q=R. Taki proces nazywa się właśnie unifikacją. Poszukiwanie takich wartości zmiennych, dla których Q=R polega na ukonkretnianiu wartości kolejnych zmiennych stałymi. Proces unifikacji zazwyczaj wiąże się z wieloma próbami podstawiania (nawrotami), gdy ukonkretnianie zmiennych pewnymi stałymi nie udaje się. Wnioskowanie w języku PROLOG polega na dowodzeniu twierdzeń (hipotez). Hipotezy dowodzi się korzystając z rezolucji, czyli przekształcania klauzul aż do doprowadzenia do powstania sprzeczności. Więcej o procesie wnioskowania w PROLOG-u będzie w późniejszych rozdziałach.

2.2. Programowanie w PROLOG-u

W założeniach swoich język PROLOG miał być językiem zupełnie innym od wszystkich dotychczas istniejących. Przede wszystkim bazuje on na innym paradygmacie programowania i zupełnie inne są jego zastosowania. Nie znaczy to, że za pomocą pewnych implementacji PROLOG-a nie da się dokonywać obliczeń ani napisać aplikacji z okienkowym interfejsem, ale

jest to dość uciążliwe i mało efektywne narzędzie o tego rodzaju funkcji. Zupełnie inaczej jest w przypadku problemów reprezentacji zależności logicznych, wnioskowania, dowodzenia twierdzeń, przeszukiwania itp. zadań. Dla tego rodzaju problemów PROLOG jest doskonałym narzędziem pozwalającym w bardzo szybki, zwięzły i spójny sposób przedstawić problem w postaci programu, przenosząc problem jego rozwiązania na komputer. Z pewnym przymrużeniem oka można stwierdzić, że PROLOG jest swego rodzaju spełnieniem marzeń programisty, bo jest to język, w którym samo opisanie problemu wystarczy aby go rozwiązać.
Niewątpliwie wiele decyzji jakie człowiek podejmuje, i do podejmowania których wykorzystuje swój intelekt, wymaga przedstawienia sobie pewnego modelu logicznego zaistniałej sytuacji. Taki model, jeśli jest wystarczająco spójny i precyzyjny, już sam w sobie jest programem i jego przełożenie do komputera jest zadaniem stosunkowo prostym. W przypadku klasycznych (proceduralnych) języków programowania, model ten jest pierwszym krokiem i dopiero w następnych krokach zaczynają się przysłowiowe „schody" i narasta „przepaść semantyczna" (termin wzięty z [17]) między modelem problemu a programem. Oczywiście tak napisanego programu nikt poza informatykiem już nie będzie w stanie zrozumieć.
Program w języku PROLOG odwzorowuje pewien fragment rzeczywistości, opisując obiekty ze świata rzeczywistego i relacje między nimi. Oczywiście owe obiekty nie mają nic wspólnego z obiektowością znaną z innych języków programowania. Nie ma tu dziedziczenia, polimorfizmu, hermetyzacji i tego wszystkiego co się wiąże z tak pojmowaną obiektowością. Obiekty które modeluje język PROLOG to dowolnego rodzaju byty jakie pojawiają się w rzeczywistości, np. stwierdzenie: „Jaś ma rower" dotyczy dwóch obiektów: Jasia i roweru. Taki obiekt nie jest strukturą danych jaka występuje np. w języku JAVA a odwzorowaniem jakiegoś bytu. Żaden obiekt nie ma wynikających ze swej natury właściwości odróżniających go od innego (poza oczywiście nazwą: „Jaś" i „rower"). Takie obiekty wchodzą w różnego rodzaju relacje, które mogą odwzorowywać relacje ze świata rzeczywistego (relacja między dwoma obiektami: „ma"). W programach języka PROLOG obiekty ze świata rzeczywistego stają się stałymi a relacje noszą nazwę predykatów. Podobnie jak w logice pierwszego rzędu predykaty mogą mieć dowolną liczbę argumentów i podobnie można je interpretować. Można też zauważyć, że predykat bez argumentów jest tak naprawdę stałą. Program w języku PROLOG składa się z:
Deklaracji faktów, które odnoszą się do obiektów i relacji.
Deklaracji zasad dotyczących obiektów i związków między nimi.
PROLOG jest językiem konwersacyjnym, czyli po kompilacji użytkownik może zadawać pytania o obiekty i związki między nimi.
Uwagi składniowe:

Składnia języka PROLOG jest bardzo prosta, co dodatkowo zwiększa łatwość tworzenia w nim programów. Do deklaracji faktów koniecznie należy wiedzieć, że:
Nazwy predykatów i stałych (które jak wcześniej zauważono, są szczególnym przypadkiem predykatu bez argumentów) pisze się małą literą. Predykat zapisuje się w takiej formie:
`nazwa_predykatu(argument_1, argument_2,...).`
W programach języka PROLOG również można wykorzystywać zmienne. Zmienne w programie nie są w żaden sposób deklarowane a komputer rozpoznaje je po tym, że, w odróżnieniu od predykatów i stałych, rozpoczynają się wielką literą.
Znak przecinka , oznacza logiczne *i*, natomiast znak średnika: ; oznacza logiczne *lub*.

2.2.1. Deklaracja faktów

Fakty deklaruje się wprowadzając nazwę predykatu i wartości argumentu w takiej formie:
`chlopak(jas).`
Powyższy fakt oznacza, że prawdą jest, że stała jas jest argumentem predykatu chlopak, czyli obiekt jas jest argumentem relacji chlopak, a interpretować go można jako deklarację, że Jaś jest chłopakiem. Nic nie stoi na przeszkodzie by stwierdzić, że Krzyś również jest chłopakiem:
`chlopak(krzys).`
Analogicznie taką deklarację:
`dziewczyna(zosia).` można interpretować jako deklarację, że Zosia jest dziewczyną. Predykat:
`lubi(jas, zosia).`
Oznacza, że zachodzi relacja „lubi" między obiektami jas i zosia, co znowu można interpretować, że Jaś lubi Zosię. Podobnie jak w logice pierwszego rzędu warto nadmienić, że z faktu, że zachodzi relacja `lubi(jas, zosia)`. Nie wynika, że zachodzi `lubi(zosia, jas)`.
Można deklarować także predykaty trzy i więcej-argumentowe, np.:
`daje(jas, zosia, kwiaty).`
Interpretacja kolejności poszczególnych argumentów (w tym przypadku: kto, komu i co daje) jest dowolna, ale ważne jest by trzymać się jednej konwencji. Można zadeklarować również sam obiekt bez wiązania go z żadną relacją.

2.2.2. Zapytania

Zadeklarowane fakty są już najprostszym programem i po jego skompilowaniu i uruchomieniu użytkownik ma możliwość konwersacji z systemem

i dokonywania zapytań. Najprostszym zapytaniem do programu złożonego z trzech deklaracji faktów z poprzedniego podrozdziału może być:

```
dziewczyna(zosia).
```

Przy takim zapytaniu system odpowie:

```
yes
```

Jeśli zapytanie będzie takie

```
dziewczyna(jas).
```

Odpowiedź będzie oczywiście taka:

```
no
```

Gdy zapyta się system o to, czy Jaś lubi Zosię, czyli:

```
lubi(jas, zosia).
```

To system odpowie:

```
yes
```

Gdy zapyta się system o to czy Zosia lubi Jasia:

```
lubi(zosia, jas).
```

Odpowiedź będzie przecząca:

```
no
```

Nic nie stoi na przeszkodzie aby zapytać o całe wyrażenie, którego składniki będą połączone znanymi funktorami, np.:

```
dziewczyna(zosia), lubi(jas, zosia).
```

Co co można zinterpretować tak: „czy Zosia jest dziewczyną i Jaś lubi Zosię?" i odpowiedź wtedy będzie taka:

```
yes
```

Gdy zaś zostanie postawione takie zapytanie:

```
chlopak(jas), lubi(zosia, jas).
```

Odpowiedź będzie przecząca:

```
no
```

Przyczyna jest oczywista: co prawda Jaś jest chłopakiem ale nigdzie nie jest zadeklarowane, że Zosia lubi Jasia. Wyrażenia są połączone spójnikiem *i* wobec czego całe wyrażenie jest fałszywe.

Użyteczność takich odpowiedzi jest niezbyt wielka ale język PROLOG daje znacznie większe możliwości, które wynikają z możliwości wykorzystania zmiennych. Gdy postawi się takie zapytanie:

```
dziewczyna(X).
```

system odpowie:

```
X = zosia.
```

Takie zapytanie można zinterpretować jako: „kto jest dziewczyną". A mechanizm wnioskujący języka będzie próbował podstawiać za X takie stałe, które spełniają ten predykat. Ponieważ w programie jest zadeklarowana tylko jedna dziewczyna, to system poda, że owym X-em jest Zosia.

Gdyby zapytać o to kogo lubi Jaś:

```
lubi(jas,X).
```

System odpowie:
```
X= zosia.
```
Ciekawa sytuacja będzie, gdy postawi się pytanie:
```
chlopak(X).
```
Ze względu na to, że wcześniej zostało zadeklarowanych dwóch chłopaków Jaś i Krzyś, to system da taką odpowiedź:
```
X= jas
```
gdy naciśnie się klawisz „;" i ENTER pojawi się:
```
X= krzys.
```
Na zapytanie: „kto jest chłopakiem" otrzymano dwie odpowiedzi (rozdzielone średnikiem, czyli logicznym *lub*), co oznacza, że chłopakiem jest zarówno Jaś jak i Krzyś. Taka odpowiedź pokazuje, że mechanizm wnioskowania wbudowany w język PROLOG daje możliwość znalezienia nie tylko jednej odpowiedzi na zadane pytanie ale pozwala uzyskać wszystkie możliwe odpowiedzi na pytanie, czyli takie podstawienia stałych za zmienne, które można potwierdzić w programie. W analizowanym przypadku zarówno stała „jas", jak i stała„krzys" była zadeklarowana jako argument predykatu „chlopak", wobec czego system mógł podstawić te stałe za zmienną „X". Stała „zosia" ani stała „kwiaty" nie była zadeklarowana jako argument predykatu „chłopak", w związku z czym nie było możliwe podstawienie jej za zmienną. Gdyby zadać zapytanie o to, czy jest jakiś chłopak, który lubi Zosię:
```
chlopak(X), lubi(X, zosia).
```
System udzieliłby odpowiedzi:
```
X= jas.
```
Jak w przypadku tego rodzaju zapytania działa mechanizm wnioskowania? Otóż system szuka stałą, która podstawiona pod X-a, spełnia pierwszy warunek (następuje ukonkretnienie zmiennej X), następnie sprawdza, czy stałą podstawiona za X (w tym wypadku „jas") można uzgodnić dla drugiego predykatu; w powyższym przypadku tak, w związku z czym system podaje odpowiedź na zapytanie. Następnie system sprawdza, czy nie ma innej możliwości podstawienia (mechanizm nawracania!), czyli sprawdza, czy jeszcze jakaś stała nie spełnia pierwszego predykatu i rzeczywiście stałą „krzys" również można podstawić za zmienną X, ale próba ukonkretnienia tej stałej dla drugiego predykatu zawodzi, w związku z czym dla tylko jednego podstawienia stałej za zmienną („jas") możliwe było uzgodnienie zapytania.
Gdyby zadać pytanie:
```
chlopak(X), lubi(zosia, X).
```
System starałby się ukonkretniać zmienną X najpierw stałą „jas", dla której wnioskowanie zawiedzie (nie jest zadeklarowane, że Zosia lubi Jasia), następnie stałą „krzys", dla której również wnioskowanie zawiedzie, w związku z czym odpowiedź systemu będzie:
```
no
```

Uwagi składniowe:
Zanim wprowadzi się pojęcie reguły znów warto podać kilka informacji składniowych, przede wszystkim ważny jest znak *:-* oznacza on „jeżeli", bądź „wtedy gdy" i używany jest w regułach do oddzielenia następnika od poprzednika.

2.2.3. Reguły

Sama deklaracja faktów nie daje zbyt wielkich możliwości wyrażania treści, pozwala jedynie deklarować prawdziwość pewnych szczególnych obiektów i relacji między nimi, nie dając żadnych możliwości deklarowania bardziej ogólnych stwierdzeń. Gdyby do poprzedniego programu dołączyć kilka nowych faktów:

```
chlopak(jas).
chlopak(krzys).
chlopak(jozef).
dziewczyna(zosia).
dziewczyna(marysia).
dziewczyna(jadzia).
lubi(jas, zosia).
lubi(jas, piwo).
lubi(jozef, piwo).
lubi(marysia, piwo).
daje(jas, zosia, kwiaty).
```

To można po kompilacji zadawać do systemu różne zapytania dotyczące relacji między obiektami. Gdyby ktoś chciał w programie wyrazić bardziej ogólną zależność, np. taką, że Krzyś lubi każdego, kto lubi piwo, to chcąc wyrazić to najprościej za pomocą deklaracji faktów trzeba by najpierw zbadać, kto lub co lubi piwo a następnie zadeklarować, że Krzyś lubi właśnie te obiekty:

```
lubi(krzys, jozef).
lubi(krzys, marysia).
```

Takie rozwiązanie jest oczywiście pozbawione jakiegokolwiek sensu, dlatego chcąc wyrazić taką treść należy posłużyć się znaną z logiki pierwszego rzędu klauzulą Horna, którą w języku prolog nazywa się po prostu regułą. Zapisać ją można tak:

```
lubi(krzys, X) :- lubi(X, piwo).
```

Po lewej stronie wyrażenia jest wniosek reguły, po prawej zaś warunek. Warunków oczywiście może być więcej i do i łączenia można używać zarówno funktora *i* jak i *lub*, jednakże funktor *lub* stosuje się rzadko. Gdyby chcieć wyrazić, że Jaś daje kwiaty każdej dziewczynie, którą lubi, to klauzula mogłaby wyglądać tak:

```
daje(jas, X, kwiaty):- lubi(jas, X), dziewczyna(X).
```

Przykład ten, podobnie jak i powyższe wygląda niewątpliwie bardzo podobnie do przykładów różnych formuł logiki pierwszego rzędu i rzeczywiście większość formuł logiki predykatów daje się prawie bezpośrednio przełożyć do języka PROLOG.

Uwagi składniowe:

Obok stałych, predykatów i zmiennych w języku PROLOG można również wykorzystywać termy złożone zwane czasem strukturami. Owe termy złożone to pojedyncze obiekty składające się z zestawu innych obiektów. Składniki struktury są pogrupowane aby ułatwić przetwarzanie.

W programach języka PROLOG można wykorzystywać jeszcze kilka innych funktorów np.:

— znak `_` jest to tzw zmienna anonimowa, która oznacza jakąś, nieistotną z punktu widzenia procesu wnioskowania wartość. Stosuje się ją wtedy, gdy w danym wnioskowaniu konkretna wartość jaką można przypisać tej zmiennej nie jest potrzebna do wnioskowania. Próbując zdefiniować jednoargumentowy predykat „rodzic", korzystając z dwuargumentowego predykatu „ojciec" można to zrobić z wykorzystaniem zmiennej anonimowej:
```
rodzic(X):- ojciec(X,_).
```
taką regułę tłumaczyć można tak, że rodzicem jest każdy, kto jest ojcem, nie jest w tym wypadku istotne, czyim jest się ojcem.

— Znak *!* jest to operator odcięcia, który nakazuje programowi aby nie uwzględniał wcześniejszych możliwości wyboru podczas nawracania. Najczęściej stosowane jest to, w celu przyspieszenia działania programu poprzez ograniczenie liczby nawrotów [7].

— Najbardziej charakterystyczną strukturą danych dla języka PROLOG jest lista. Lista ma taką postać: `[a1, a2, a3, ... , an]`. Elementami listy mogą być stałe, zmienne, struktury oraz inne listy. Elementy listy rozdzielone są przecinkami. Listę można również zapisać w takiej postaci:
```
[G|O]
```
gdzie: „G" to głowa, czyli pierwszy element listy a „O" to ogon, czyli cała reszta listy. Listę można również przedstawić tak:
```
[G1,G2|O]
```
Gdzie G1 i G2 to dwa pierwsze elementy listy, O to ogon listy. Listę pustą przedstawia się tak: `[]`

— W język PROLOG jest szereg wbudowanych predykatów, tzw. predykatów standardowych. Predykaty te mają różne zastosowanie: część z nich np. odpowiada za komunikację ze światem zewnętrznym, czyli operacje wejścia i wyjścia, część pozwala na tworzenie interfejsów graficznych itp. Predykaty standardowe często różnią się w różnych wersjach języka.

Jednym z ciekawszych predykatów często wykorzystywanym w procesie wnioskowania jest bezargumentowy predykat „fail", który oznacza (jakkolwiek by to dziwnie nie zabrzmiało) fakt zawsze nieprawdziwy i bywa wykorzystywany z operatorem odcięcia

— Operator równości = w przypadku, gdy w wyrażeniu występują zmienne próbuje dokonać takiego ich ukonkretnienia aby wyrażenia po obu stronach były takie same, czyli jeśli:
`X= lubi(jas, zosia).`
to pod X ukonkretnia się struktura `lubi(jas, zosia)`
Generalnie język PROLOG nie jest przeznaczony do prowadzenia obliczeń, jednakże pewne proste operacje arytmetyczne są możliwe, operatory +, -, *, / i inne są tak naprawdę predykatami wbudowanymi, które mogą mieć formę prefixową, jak inne predykaty: `+(1,2)` lub formę infixową: `1+2`. Warto jednak zauważyć, że dla PROLOG-a wyrażenie:
`1+2 = 3`
jest fałszem, ponieważ operator równości (który jest operatorem infixowym) dokonuje porównania symbolicznego. Jeśli w danej klauzuli konieczne jest porównanie arytmetyczne stosuje się operator (infixowy predykat) *is*, czyli:
`1+2 is 3`
Chcąc porównać liczby można wykorzystać szereg operatorów porównania:
— `X =:= Y` – X i Y są tą samą liczbą,
— `X =\= Y` – X i Y są różnymi liczbami,
— `X < Y` – X jest mniejsze od Y,
— `X > Y` – X jest większe od Y,
— `X =< Y` – X jest mniejsze lub równe Y,
— `X >= Y` – X jest większe lub równe Y,

Argumentami tych operatorów mogą być zmienne pod które ukonkretnione mogą być liczby całkowite, liczbami zapisanymi jako stałymi bądź całe wyrażenia.

2.2.4. Przykładowe problemy i ich rozwiązania w języku PROLOG

Język PROLOG jest językiem, który jest szczególnie predysponowany do modelowania zależności logicznych i jako taki znakomicie się nadaje do budowy, bądź modelowania wszystkich tych systemów sztucznej inteligencji, których zasada działania opiera się na reprezentacji wiedzy i wnioskowaniu. Poniżej zostanie zaprezentowanych kilka różnych przykładów prostych zadań wraz z ich rozwiązaniami w języku PROLOG, część z nich będzie uproszczonymi wersjami problemów o charakterze komercyjnym a część bę-

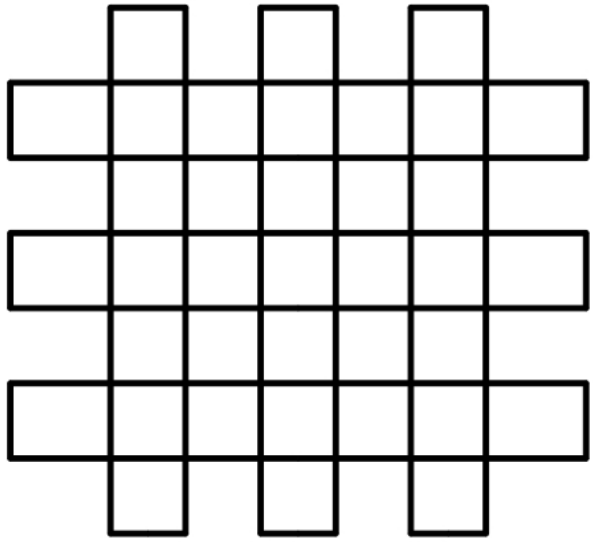
dzie miało formę zagadek logicznych. Ponieważ najlepszą metodą nauki są ćwiczenia praktyczne, dlatego autor zachęca czytelnika się aby próbował samodzielnie uruchomić opisane tu przykłady programów.

Krzyżówka

Pierwsze zadanie będzie miało formę krzyżówki. Zadanie owo zostało znalezione w internecie (http://www.brics.dk/ brabrand/dProgSprog/exercises.html) i brzmi tak: Sześć siedmioliterowych słów należy tak wpisać do poniżej zaprezentowanej krzyżówki aby litery na przecięciach słów się zgadzały. Słowa te to:

— abandon
— anagram
— abalone
— enhance
— elegant
— connect

Krzyżówka wygląda tak: Pojawia się w takiej sytuacji oczywiste pytanie:



Rysunek 2.1. Krzyżówka.

od czego zacząć? Program języku PROLOG jest przede wszystkim modelem problemu, czyli pierwszym (głównym i zazwyczaj jedynym) zadaniem programisty jest zamodelowanie problemu za pomocą formalizmów logiki pierwszego rzędu, bądź wprost klauzul języka PROLOG. Aby zamodelować powyższe zadanie należy na wstępie skupić się na problemie definicji faktów, czyli należy określić co wiadomo z samych założeń. Przede wszystkim

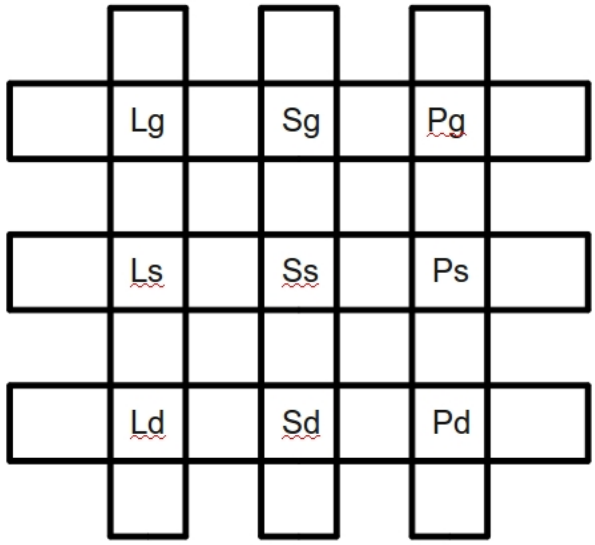
należy zadeklarować słowa jakie konieczne będą do rozwiązania krzyżówki. Ponieważ do rozwiązania krzyżówki konieczne jest nie tylko całe słowo ale także poszczególne litery przydatne będzie zdefiniowanie predykatu „slowo", którego pierwszym argumentem może być całe słowo, kolejne siedem argumentów będzie poszczególnymi literami słowa, w takiej postaci:

Listing 2.1. Definicje predykatu slowo

```
slowo(abalone,a,b,a,l,o,n,e).
slowo(abandon,a,b,a,n,d,o,n).
slowo(enhance,e,n,h,a,n,c,e).
slowo(anagram,a,n,a,g,r,a,m).
slowo(connect,c,o,n,n,e,c,t).
slowo(elegant,e,l,e,g,a,n,t).
```

Podobnie jak w przykładach omawianych wcześniej kolejność argumentów nie ma znaczenia dla omawianego problemu, ważne tylko by zachować konsekwencję. Można również posilić się na redukcję liczby argumentów tzn. wywnioskować pełną nazwę słowa z jego liter ale wiąże się to z dodatkowym komplikowaniem programu a nie wnosi wiele dla analizowanego problemu. Gdy poszczególne słowa zostały już zdefiniowane, kolejnym krokiem budowy modelu i zarazem programu jest zdefiniowanie samej krzyżówki. Krzyżówka składa się z sześciu siedmioliterowych słów, trzy z nich są ułożone wertykalnie, trzy z nich horyzontalnie, nic nie stoi na przeszkodzie aby przypisać tym słowom zmienne np.: H1, H2, H3, dla trzech słów położonych horyzontalnie (w kolejności góra – dół) i V1, V2, V3, dla trzech słów ułożonych wertykalnie (od lewej do prawej). Główna trudność właściwego ułożenia słów w krzyżówce polega na takim ich dobraniu aby litery na skrzyżowaniach poszczególnych wyrazów były takie same, czyli np. druga litera wyrazu V1 musi być taka sama jak druga litera wyrazu H1 itd. Model krzyżówki w związku z tym opierał się będzie na 6 zmiennych: H1, H2, H3, V1, V2, V3, pod które będą ukonkretniane poszczególne słowa. Poza tym kolejne 9 zmiennych odpowiadać będzie za litery „leżące" na skrzyżowaniach słów, w ten sposób:

Zmienna Lg odpowiada drugiej literze słowa ukonkretnionego pod zmienną H1 i V1, natomiast zmienna Pg odpowiada drugiej literze słowa ukonkretnionego pod V3 i szóstej literze słowa ukonkretnionego pod zmienną H1 itd. Warunki poprawnego rozwiązania są wobec tego takie, że poszczególne słowa muszą zostać przypisane zmiennym od V1 do H3 a poszczególne litery tych słów na skrzyżowaniach wyrazów powinny się zgadzać. W związku z powyższym definicja predykatu „krzyzowka" może wyglądać tak:



Rysunek 2.2. Krzyżówka z nazwami zmiennych.

Listing 2.2. Definicja predykatu krzyzowka

```
krzyzowka(V1,V2,V3,H1,H2,H3) :- slowo(V1,_,Lg,_,Ls,_,Ld,_),
    slowo(V2,_,Sg,_,Ss,_,Sd,_),slowo(V3,_,Pg,_,Ps,_,Pd,_),
    slowo(H1,_,Lg,_,Sg,_,Pg,_), slowo(H2,_,Ls,_,Ss,_,Ps,_),
    slowo(H3,_,Ld,_,Sd,_,Pd,_).
```

Ponieważ w krzyżówce jest sześć wyrazów, to w definicji predykatu sześć razy wykorzystano predykat „slowo". Wyrazy w krzyżówce przecinają się w drugiej, czwartej i szóstej literze, w związku z czym przy każdym predykacie „slowo" zaznaczono właściwe zmienne odpowiadające poszczególnym literom odpowiedniego słowa, np. druga litera w słowie H1 i V1 powinna być taka sama i rzeczywiści w obu tych słowach oznacza ją jedna zmienna Lg. Litery, które dla rozwiązania nie są istotne ukryte są pod znakiem zmiennej anonimowej.

Rozwiązanie powyższego zadania sprowadza się do przeszukania przestrzeni możliwych układów wyrazów w krzyżówce i znalezienia takich, dla których spełnione będą wszystkie ograniczenia zdefiniowane w predykacie „krzyzowka". Program w języku PROLOG nie zawiera żadnych wskazówek jak problem ma być rozwiązany. Problemem tym zajmuje się sam kompilator. Sposób rozwiązania tego zadania oraz algorytm jakim posługują się programy w języku PROLOG zostanie zaprezentowany później.

Bardzo wiele problemów sztucznej inteligencji sprowadza się do problemu przeszukiwania. Ponieważ, mimo nieustannego rozwoju informatyki, kom-

puter ciągle nie potrafi nic „wymyślić", „wynaleźć„ i „wpaść" na rozwiązanie dlatego wszystkie problemy, rozwiązanie których trzeba właśnie wymyślić są rozwiązywane na zasadzie przeszukiwania przestrzeni zdarzeń. Zazwyczaj przestrzeń ta jest zbyt wielka by było możliwe proste i szybkie przeszukanie jej np. przeglądem zupełnym w związku z czym zazwyczaj konieczne jest wykorzystanie jakiegoś nieco bardziej zaawansowanego algorytmu bądź heurystyki przeszukiwania.

Rekurencja

W programach języka PROLOG bardzo często wykorzystuje się rekurencję. Stosowana ona jest zazwyczaj do problemów, w których nie jest możliwe określenie z góry długości ścieżki wnioskowania bądź przeszukiwania. Najprostszym przykładem takiego zadania jest definicja przodka, dla takiego zbioru faktów:

Listing 2.3. Relacje rodzinne

```
rodzic(jas, jozek).
rodzic(jozek, franek).
rodzic(franek, marek).
rodzic(marek, jacek).
rodzic(jacek, jerzy).
```

Ponieważ przodkiem jest rodzic to najprościej będzie zdefiniować to tak:

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

Przodkiem również jest dziadek, wobec czego:

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

Czyli aby „X" był przodkiem „Y" musi być rodzicem jakiegoś „Z", który to „Z" musi być rodzicem „Y". Przodkiem jest także pradziadek, czyli:

```
przodek(X,Y):- rodzic(X, Z), rodzic(Z, K), rodzic(K, Y).
```

Można oczywiście w ten sposób dodawać kolejne generacje przodków aż do Adama i Ewy, ale nie da się zaprzeczyć, że jest to rozwiązanie mało efektywne. Ratunkiem w takiej sytuacji może być właśnie rekurencja, wg której przodkiem jest rodzic lub rodzic przodka, czyli:

Listing 2.4. Definicja przodka

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

W takiej sytuacji stawiając skompilowanemu programowi zapytanie:
`przodek(jas, franek).`
System odpowie:
`yes`
Procedura jego wnioskowania będzie wyglądała tak:
Wg pierwszej definicji przodkiem jest rodzic, czyli za zmienną X podstawiona zostanie stała „jas", za zmienną Y stała „franek", ponieważ tak postawionego zapytania nie da się uzgodnić (nigdzie nie jest zadeklarowane, że Jaś jest rodzicem Franka) system cofnie się i wykorzysta drugą definicję przodka, podstawiając za zmienną Z taką stałą, by prawdą było, że Jaś jest „jej" rodzicem, następnie zaś system spróbuje sprawdzić, czy stała ukonkretniona pod zmienną Z jest przodkiem franka, a sprawdzić to może sprawdzając, czy jest ona rodzicem franka (wg pierwszej definicji przodka).

Programy bazujące na rekurencji mają pewną bardzo charakterystyczną budowę (będzie to również widoczne w następnych przykładach), otóż praktycznie zawsze są dwie główne definicje predykatu modelującego problem (czasem, dla bardziej skomplikowanych zagadnień może być ich więcej ale też dadzą się one sprowadzić do kilku grup po dwie definicje) i pierwsza z nich pokazuje najprostszą możliwą sytuację, np.: przodkiem jest rodzic, następnie druga definicja opisuje sytuacje bardziej skomplikowane wywołując rekurencyjnie siebie samą (np. przodkiem jest rodzic przodka).

Innym ciekawym zadaniem nieco bliższym praktycznym i komercyjnym zastosowaniom jest zadanie poszukiwania połączenia kolejowymi między różnymi miastami dysponując listą połączeń bezpośrednich. Cały problem takiego zadania polega na tym że trudno jest z góry przewidzieć skąd i dokąd podróżny będzie chciał jechać, w związku z czym nie wiadomo ile (jakich) przesiadek będzie musiał wykonać aby dojechać z jednego miasta do drugiego. Najprościej bezpośrednie połączenie między miastami można zdefiniować za pomocą dwuargumentowego predykatu „polaczenie":

Listing 2.5. Lista bezpośrednich połączeń

```
polaczenie(lublin, radom).
polaczenie(radom, katowice).
polaczenie(katowice, wroclaw).
polaczenie(wroclaw, poznan).
polaczenie(radom, warszawa).
polaczenie(warszawa, poznan).
```

Chcą sprawdzić, czy można koleją dojechać z jednego miasta do drugiego należy zdefiniować dwuargumentowy predykat o nazwie np. „podroz", któ-

rego pierwszy argument będzie miastem startu, drugi zaś celem podróży. Ponieważ nie wiadomo ile przesiadek będzie konieczne, by dojechać do celu znów konieczne będzie skorzystanie z rekurencji. Najprostsza sytuacja jest wtedy, gdy szukane połączenie jest bezpośrednie, wtedy wystarczy taka definicja:

```
podroz(X, Y):- polaczenie(X,Y).
```

Gdy nie ma bezpośredniego połączenia, wtedy konieczne jest dodanie drugiej definicji predykatu „podroz" i wykorzystanie rekurencji:

```
podroz(X, Y):- polaczenie(X, Z), podroz(Z, Y).
```

Niestety z tak sformułowanych predykatów nie można się dowiedzieć nic więcej, niż tylko, że między takim a takim miastem jest połączenie. Gdyby chcieć się dowiedzieć, w jakich miastach byłaby przesiadka wtedy trzeba posłużyć się predykatem standardowym „write", który wypisuje na ekranie zawartość argumentu.

Listing 2.6. Definicja predykatu podroz

```
podroz(X, Y):- polaczenie(X,Y), write(Y).
podroz(X, Y):- polaczenie(X, Z), podroz(Z, Y), write(Z).
```

W drugiej definicji predykatu „podroz" (pierwsza jest trywialna) oprócz zmiennych X i Y, pod które ukonkretniane są punkt wyjazdu i cel podróży występują zmienne:

— StartP pod którą ukonkretnia się godzinę rozpoczęcia podróży,
— KoniecP, pod którą ukonkretnia się godzina dojechania do ostatecznego celu podróży,
— Start, pod którą ukonkretnia się godzinę startu kolejnej przesiadki,
— Koniec, pod którą ukonkretnia się godzinę dojazdu do stacji przesiadki.

Po kompilacji takiego programu można postawić zapytanie:

```
podroz(lublin, poznan, X, Y).
```

otrzyma się wtedy takie odpowiedzi:

```
radomkatowicewroclaw
X = 8,
Y = 22 ;
poznanwarszawa
X = 8,
Y = 15 ;
poznan
false.
```

System podaje trasę (miasta przesiadek – dzięki wykorzystaniu predykatu standardowego „write") oraz godzinę wyjazdu oraz przyjazdu na stację końcową dla dwóch możliwości dojazdu.

Listy i rekurencja

Najczęściej wykorzystywaną w języku PROLOG stukturą danych są listy. Listy w połączeniu z rekurencją są bardzo mocnym narzędziem pozwalającym modelować wiele rzeczywistych problemów. Najprostszym przykładem zadania z listami i rekurencją jest sprawdzenie, czy dana stała jest częścią listy:

Listing 2.7. Program sprawdzający czy argument jest elementem listy

```
element(X,[G|_]):- X=G.
element(X,[_|O]):- element(X,O).
```

Program działa w ten sposób, że na początku sprawdza, czy dany element jest głową listy (porównuje zmienną X ze zmienną G, pod którą w tym wypadku ukonkretniona jest głowa listy), następnie, jeśli element nie jest głową listy, to sprawdza (w drugiej definicji predykatu) czy element nie jest częścią ogona, wywołując rekurencyjnie jeszcze raz predykat „element". Cała idea opiera się na takim „obieraniu" listy z poszczególnych elementów od pierwszego aż do ostatniego. Gdyby w liście nie było szukanego elementu odpowiedź byłaby:

```
no.
```

Inne, nieco ciekawsze, zadanie wykorzystujące rekurencję polega na tłumaczeniu listy słów z jednego języka na drugi, przykład będzie zawężony do 10 liczebników ale bez trudu można go rozszerzyć na większy zakres słów. Na wstępie analogicznie jak było w kilku poprzednich przykładach należy zadeklarować predykaty opisujące najprostsze zależności: w tym wypadku tłumaczenia poszczególnych liczebników:

Listing 2.8. Predykaty tłumaczące liczebniki

```
tlumacz(jeden,one).
tlumacz(dwa,two).
tlumacz(trzy,three).
tlumacz(cztery,four).
tlumacz(piec,five).
tlumacz(szesc,six).
tlumacz(siedem,seven).
tlumacz(osiem,eight).
tlumacz(dziewiec,nine).
tlumacz(zero,zero).
```

Dysponując słownikiem poszczególnych słów można przystąpić do definicji predykatu tłumaczącego listę liczebników. Najprostsza sytuacja jest wtedy, gdy lista jest pusta, wtedy odpowiedzią jest również lista pusta. Gdy lista nie

jest pusta, wtedy predykat działa tak, że z obu list (źródłowej i wynikowej) bierze głowy, za pomocą predykatu tlumacz ukonkretnia brakujące wartości zmiennych, następnie tłumaczy ogony listy rekurencyjnie wywołując predykat „tlumaczliste":

Listing 2.9. Predykaty tłumaczący listy liczebników

```
tlumaczliste([],[]).
tlumaczliste([We|O1],[Wy|O2]):- tlumacz(We,Wy),
    tlumaczliste(O1,O2).
```

Po kompilacji i zadaniu zapytania:
`tlumaczliste([jeden, trzy, osiem], X).`
otrzyma się:
`X= [one, three, eight].`
Nic nie stoi na przeszkodzie aby zapytać o tłumaczenie z j. angielskiego na polski:
`tlumaczliste(X, [one, nine, seven]).`
otrzyma się:
`X= [jeden, dziewiec, siedem].`
Takie tłumaczenie to oczywiście najprostsze możliwe rozwiązanie i przekładanie całych wyrażeń języka naturalnego jest pozbawione większego sensu, gdyż taki program nie uwzględnia ani gramatyki ani kontekstu wypowiedzi, które są niezbędne do uzyskania poprawnego tłumaczenia. Pomimo tego, przykład ten pokazuje, że stosunkowo łatwo można w języku PROLOG realizować systemy przetwarzania języka naturalnego, także takie bardziej skomplikowane, ujmujące już w sobie rozbiór zdania, wykrywanie części mowy itp. Ze względu na to, iż PROLOG bardzo mocno jest osadzony w logice, to wszelkiego rodzaju zadania rozbioru logicznego zdania, wyróżniania w nim konkretnych części (np. frazy rzeczownikowe, czasownikowe, podmiot, orzeczenie itp.) są w nim znacznie prostsze do realizacji niż w klasycznych językach proceduralnych. Również manipulacja tekstem naturalnym w postaci np. tłumaczenia z języka naturalnego na język logiki jest również znacznie prostsza niż w językach proceduralnych. Właśnie podczas przetwarzania zdań języka naturalnego szczególnie wyraźnie widoczne jest to, że wyrażenia z języka naturalnego często nie różnią się bardzo od ich zapisu w języku logiki, dzięki czemu język PROLOG pozwala na stosunkowo proste i czytelne modelowanie procesów ludzkiego rozumowania. Generalnie przetwarzanie języka naturalnego jest zadaniem niezwykle skomplikowanym i próby praktycznej realizacji narzędzi bazujących na nim są dalekie od ideału. Nie można jednak zaprzeczyć, że potencjalne możliwości wykorzystania są olbrzymie: począwszy od systemów konwersacji dla niepełnosprawnych, narzędzi edukacyjnych, skończywszy na automatycznym streszczaniu doku-

mentów i automatycznym tłumaczeniu tekstów w różnych językach. Trudności powstają nie tylko podczas rozbioru zdania i tłumaczenia słów: znacznie trudniejszy do analizy jest problem rozumienia kontekstu wypowiedzi: gdyż to samo zdanie może mieć w zależności od właśnie kontekstu sytuacji zupełnie różne znaczenie np.: Jeśli student na egzaminie błyszczał wiedzą i egzaminator skomentował jego wypowiedź: „Bardzo dobrze Pan się dziś nauczył" to takie zdanie można zinterpretować jako wyraz podziwu, gdyby zaś ów student nie odpowiedział na żadne pytanie a egzaminator skomentował by to dokładnie w takich samych słowach byłby to wyraz raczej szyderstwa i dezaprobaty. Język logiki jest w związku z tym o tyle użyteczny o ile można założyć, że wypowiedzi w języku naturalnym przestrzegają zasad logiki i, ponieważ wiadomo, że nie zawsze tak jest, dlatego jest to tylko jedno z kilku niedoskonałych narzędzi, ułatwiających realizację narzędzi bazujących na przetwarzaniu języka naturalnego.

Podstawowa trudność w pisaniu programów języka PROLOG nie leży w wymyśleniu i implementacji algorytmu, a w odpowiednio precyzyjnej definicji zadania. Ponieważ nie trzeba szczegółowo opisywać algorytmu programy są zwięzłe i stosunkowo czytelne. Przy skomplikowanych zadaniach problem polega jednak przede wszystkim na „wpadnięciu" na pomysł jak je zdefiniować. Poniżej zostanie przedstawiony przykład zadania, które wymaga właśnie pomysłu na to jak go przedstawić aby wyrazić wszystkie zależności między obiektami. Zadanie to jest uproszczoną wersją zagadki Carolla (lub Einsteina – bo różne istnieją wersje) i brzmi tak:

Przy ulicy stoją trzy domy: każdy z nich ma inny kolor: jeden jest czerwony, drugi zielony a trzeci niebieski. W każdym z tych domów mieszka osoba innej narodowości: w jednym Anglik, w drugim Hiszpan a w trzecim Japończyk. Każdy z nich hoduje jakieś zwierzę: jedna osoba ma ślimaka, druga jaguara a trzecia zebrę. Wiadomo jeszcze, że Anglik mieszka w czerwonym domu, Hiszpan ma jaguara, Japończyk mieszka na prawo od właściciela ślimaka, a właściciel ślimaka mieszka na lewo od niebieskiego domu. Pytanie brzmi: Kto hoduje zebrę?

Rozwiązanie tej zagadki w dużej mierze opiera się na znalezieniu odpowiedniej struktury danych, która pozwoliłaby nie tylko wyrazić to, kto w którym domu mieszka ale także relacje „na prawo od" i „na lewo od". Implementacja rozwiązania w klasycznym proceduralnym języku programowania jest dość pracochłonna i wymaga dodatkowo implementacji algorytmu, który przeszukałby przestrzeń układów mieszkańców i kolorów tych domów. Najprostszy jest oczywiście w takiej sytuacji przegląd zupełny, który sprawdza wszystkie możliwe stany. Niestety jak to zwykle bywa al-

gorytm najprostszy jest najmniej efektywny: Dla tak prostego zadania jest 216 możliwych stanów do sprawdzenia, gdzie dla pełnej wersji tej zagadki (5 domów i pięć cech do spełniania) jest ich aż 24883200000. Zrozumiałe jest, że do takiego zadania lepiej jest wykorzystać algorytm bardziej skomplikowany ale pozwalający na znaczną redukcję stanów do sprawdzenia. Próbując rozwiązać to zadanie wykorzystując język PROLOG nie ma konieczności implementacji algorytmu, bo z tym poradzi sobie kompilator (to jak są rozwiązywane tego rodzaju zadania w PROLOG-u i na czym polega ich skuteczność zostanie opisane później), natomiast ważne jest by w sposób właściwy opisać obiekty i zależność w zagadce.

Zadanie to można zrealizować na kilka sposobów poniżej zostanie zaprezentowany jeden z nich. Pierwszym krokiem powinno być zdefiniowanie sposobu zakodowania istniejących zależności. Na pierwszy rzut oka wydaje się, że należałoby zdefiniować predykaty opisujące relacje między obiektami ale można zadanie to rozwiązać na inny sposób. Opiera się on na spostrzeżeniu, że w zagadce relacje można uprościć do takiej postaci: „występują razem", „na lewo", „na prawo", czyli jeśli człowiek mieszka w którymś domu to człowiek i dom występują razem, jeśli człowiek hoduje któreś zwierze, to też oni występują razem, jeśli człowiek mieszka na prawo od hodowcy ślimaka tzn. że między ślimakiem a tym człowiekiem jest relacja „na prawo". Kolejne spostrzeżenie wiąże się z tym, że skoro w zagadce są relacje na lewo i na prawo to znaczy, że domy są ustawione w jednej linii, w związku z czym można je, na użytek rozwiązania, ponumerować i oznaczyć tak, że dom na prawo ma numer większy od numeru na lewo. Skoro domy zostały ponumerowane, to numery te można również podstawić pod ich mieszkańców, czyli jeśli dom o takim a nie innym kolorze ma numer jeden to mieszkańcom tego domu (człowiekowi i zwierzęciu) można też ten numer przypisać. Mając takie założenia można znów zredukować liczbę relacji i zauważyć w takim razie, że nie są konieczne relacje „występują razem", „na lewo", i „na prawo" ponieważ wystarczy porównać numery domu przypisane każdemu obiektowi. W związku z tym zamiast definiować stałe odpowiadające poszczególnym obiektom można założyć, że każdy z obiektów będzie opisany przez jedną zmienną (Anglik, Hiszpan, Japonczyk, Czerwony, Zielony, Niebieski, Zebra, Slimak, Jaguar) a wartością każdej zmiennej będzie numer domu.
Wystarczy, wobec tego, zdefiniować zakres wartości jaki te zmienne mogą przyjmować (predykat „dom(numer domu)") i można definiować zależności:

Listing 2.10. Rozwiązanie zagadki Einsteina

```
1 dom(1).
  dom(2).
```

```
3 dom(3).
  zebra(Zebra, Jaguar ,Slimak,Czerwony, Niebieski, Zielony,
5           Anglik, Japonczyk, Hiszpan):-
      dom(Zebra),
7     dom(Jaguar),
      dom(Slimak),
9     dom(Czerwony),
      dom(Niebieski),
11    dom(Zielony),
      dom(Anglik),
13    dom(Japonczyk),
      dom(Hiszpan),
15    Zebra =\= Jaguar,
      Zebra =\= Slimak,
17    Jaguar =\= Slimak,
      Czerwony =\= Niebieski,
19    Czerwony =\= Zielony,
      Zielony =\= Niebieski,
21    Anglik =\= Japonczyk,
      Anglik =\= Hiszpan,
23    Japonczyk =\= Hiszpan,
      Czerwony = Anglik,
25    Jaguar = Hiszpan,
      Japonczyk > Slimak,
27    Slimak < Niebieski.
```

Jak działa powyższy program? Pierwsze trzy definicje deklarują, że jednoargumentowy predykat „dom" spełniony będzie dla trzech wartości argumentu (1,2,3). Określa się w ten sposób zakres wartości jaki może mieć numer domu. Dziewięcioargumentowy predykat „zebra" jest właściwym opisem problemu. Dziewięć zmiennych oznacza tu wszystkie obiekty jakie występują w zagadce tzn.: Anglika, Hiszpana, Japończyka, czerwony dom, niebieski dom, zielony dom oraz jaguara, ślimaka i zebrę. Wartość z jaką ukonkretnione będą te zmienne będzie oznaczała numer domu, który dany obiekt ma lub w nim mieszka. Pierwsze dziewięć warunków określa zakres wartości wszystkich zmiennych. Kolejne trzy warunki zastrzegają, że każde zwierze mieszka w innym domu (numery domów przypisane pod zmienne muszą być różne), kolejne trzy zastrzegają, że dom o każdym kolorze ma inny numer a kolejne trzy warunki zastrzegają, że każda osoba mieszka w innym domu. W warunku:

```
Czerwony = Anglik,
```

zastrzega się, że w czerwonym domu mieszka Anglik (wartość podstawiona pod kolor domu, czyli jego numer jest równa wartości podstawionej pod zmienną Anglik). W warunku:

```
Jaguar = Hiszpan,
```

zastrzega się, że Hiszpan ma jaguara, natomiast ostatnie dwa warunki infor-

mują, że Japończyk mieszka na prawo od ślimaka (numer domu Japończyka jest większy od numeru domu, w którym mieszka ślimak) i ślimak mieszka na lewo od niebieskiego domu.

Taka definicja zagadki jest wystarczająca i opisuje wszystkie zależności jakie w zagadce występują. Po zadaniu zapytania:
`?- zebra(Zebra, Jaguar, Slimak, Czerwony, Zielony, Niebieski, Anglik, Japonczyk, Hiszpan).` System dał kilka możliwych odpowiedzi, oto jedna z nich:

```
Zebra = 2,
Jaguar = 3,
Slimak = 1,
Czerwony = 1,
Zielony = 2,
Niebieski = 3,
Anglik = 1,
Japonczyk = 2,
Hiszpan = 3 ;
```

Wynika z niej, że zebrę ma Japończyk mieszkający w zielonym domu.

Warto w tym punkcie przyjrzeć się procesowi wnioskowania jaki jest wykonywany w programach języka PROLOG. Programista tworząc program w PROLOG-u nie wprowadza algorytmu rozwiązania problemu a jedynie jego opis wyrażony za pomocą zależności logicznych. Ponieważ program jest wyrażony za pomocą zależności logicznych a nie procedur to jego działanie polega na wnioskowaniu, czyli wyprowadzaniu nowych faktów z już istniejących, a nie, jak w klasycznych językach, na wykonywaniu poszczególnych poleceń. Podczas kompilacji system łączy program z mechanizmem wnioskowania i po jego uruchomieniu w trybie konwersacyjnym możliwe jest zadawanie pytań do systemu. Bardziej rozbudowane kompilatory pozwalają na tworzenie graficznego interfejsu wykorzystując do tego bogaty zestaw predykatów standardowych.

Na początku język PROLOG był językiem interpretowanym, dopiero znacznie później zaczęto robić kompilatory, które dziś są standardem. Wiele problemów rozwiązywanych za pomocą języka PROLOG sprowadza się do procesu przeszukiwania i rzeczywiście wnioskowanie w tym języku polega na poszukiwaniu takich stałych dla których spełnione będzie postawione zapytanie. Algorytm wykorzystany w języku PROLOG nosi nazwę standard backtracking. Zasada jego działania będzie przedstawiona na przykładzie powyższego zadania, czyli uproszczonej zagadki Einsteina. Algorytm standard bakctracking, który jest wykorzystywany przy przeszukiwaniu w ję-

zyku PROLOG działa na zasadzie ukonkretniania wartości poszczególnych zmiennych stałymi, jeśli dla danej stałej nie jest możliwe ukonkretnienie, następuje cofnięcie się procesu wnioskowania (backtracking) i próba ukonkretnienia kolejnej stałej, Jeśli nie ma już stałych do podstawienia proces cofa się do wcześniej ukonkretnionej zmiennej i próbuje podstawić kolejną stałą za tą zmienną.
W przypadku powyższego programu jest zadeklarowanych 9 zmiennych (Zebra, Jaguar, Slimak, Czerwony, Zielony, Niebieski, Anglik, Japonczyk, Hiszpan) i 2 predykaty (jednoargumentowy „dom" i dziewięcioargumentowy „zebra". Gdy postawi się takie zapytanie:

```
?- zebra(Zebra, Jaguar, Slimak, Czerwony, Zielony, Niebieski,
Anglik, Japonczyk, Hiszpan).
```

System będzie usiłował ukonkretnić wartości wszystkich zmiennych, czyli wstawić takie ich wartości, dla których będą spełnione wszystkie warunki. System może podstawiać tylko takie wartości jakie zostały wcześniej zadeklarowane jako argumenty użytych w programie predykatów. Po postawieniu powyższego zapytania system rozpocznie proces ukonkretniania poszczególnych zmiennych: Predykat „zebra" będzie spełniony, gdy wartości wszystkich zmiennych będą spełniały predykat dom (pierwsze 9 warunków), w związku z czym każda z nich może przyjąć wartość 1,2, bądź 3. W kolejnym kroku system próbuje ukonkretnić pierwszą zmienną, czyli Zebra podstawiając pod nią pierwszą dostępną wartość, czyli „1", ponieważ po ukonkretnieniu nie zostaje naruszony żaden warunek (inne zmienne nie są jeszcze ukonkretnione), to system próbuje podstawić wartość za kolejną zmienną, czyli Jaguar. Pierwszą dostępną wartością jest 1 i system ją podstawia i sprawdza, czy został naruszony jakiś warunek.

```
(Zebra=1, Jaguar=1 ,Slimak,Czerwony, Niebieski, Zielony, Anglik,
Japonczyk, Hiszpan)
```

Ponieważ naruszony jest warunek, który mówi, że wartość zmiennej Zebra musi być różna od wartości zmiennej Jaguar, to podstawienie zawodzi i system próbuje ukonkretnić kolejną wartość pod zmienną Jaguar, czyli „2": `(Zebra=1, Jaguar=2 ,Slimak,Czerwony, Niebieski, Zielony, Anglik,`
`Japonczyk, Hiszpan)`
To podstawienie na razie nie narusza żadnego warunku, w związku z czym system próbuje ukonkretnić kolejną zmienną, czyli Slimak pierwszą dostępną wartością, czyli „1":

```
(Zebra=1, Jaguar=2 ,Slimak=1,Czerwony, Niebieski, Zielony, Anglik,
Japonczyk, Hiszpan)
```

Takie ukonkretnienie znów zawodzi (nie może być równa z Zebrą), w związku z czym podstawia za zmienną Slimak kolejną wartość, czyli „2":

```
(Zebra=1, Jaguar=2 ,Slimak=2,Czerwony, Niebieski, Zielony, Anglik,
```

```
Japonczyk, Hiszpan)
```
i znowu jeden z warunków zostaje naruszony (wartość zmiennej nie może być równa wartości zmiennej Jaguar), w związku z czym podstawiana jest kolejna (ostatnia) dostępna wartość, czyli „3”:
```
(Zebra=1, Jaguar=2 ,Slimak=3,Czerwony, Niebieski, Zielony, Anglik,
Japonczyk, Hiszpan)
```
W tym punkcie żadne ograniczenie nie jest naruszone i system próbuje ukonkretniać kolejną zmienną, czyli Czerwony podstawiając pod nią pierwszą dostępną wartość, czyli „1”:
```
(Zebra=1, Jaguar=2 ,Slimak=3,Czerwony=1, Niebieski, Zielony,
Anglik, Japonczyk, Hiszpan)
```
Ponieważ dla takiego ustawienia nie jest naruszony żaden warunek, to można podstawić wartość za kolejną zmienną, którą w tym wypadku jest Niebieski, pierwsza próba podstawienia:
```
(Zebra=1, Jaguar=2 ,Slimak=3,Czerwony=1, Niebieski=1, Zielony,
Anglik, Japonczyk, Hiszpan)
```
skutkuje naruszeniem warunku mówiącego, że wartość zmiennej Czerwony musi być różna od wartości zmiennej Niebieski, wobec czego system się cofa i próbuj podstawić kolejną wartość, czyli „2”:
```
(Zebra=1, Jaguar=2 ,Slimak=3,Czerwony=1, Niebieski=2, Zielony,
Anglik, Japonczyk, Hiszpan)
```
Taki układ narusza warunek mówiący, że wartość zmiennej Niebieski musi być większa od wartości zmiennej Slimak (ostatni warunek predykatu „zebra”). System próbuje ukonkretnić kolejną wartość pod zmienną Niebieski, czyli „3”:
```
(Zebra=1, Jaguar=2 ,Slimak=3,Czerwony=1, Niebieski=3, Zielony,
Anglik, Japonczyk, Hiszpan)
```
Niestety w takim ustawieniu również naruszony jest ostatni warunek, w związku z czym system nie ma więcej wartości, które mógłby podstawić pod zmienną Niebieski, dlatego następuje cofnięcie procesu i próba podstawienia kolejnej wartości pod zmienną Czerwony:
```
(Zebra=1, Jaguar=2 ,Slimak=3,Czerwony=2, Niebieski, Zielony,
Anglik, Japonczyk, Hiszpan)
```
Nie jest naruszony żaden warunek, dlatego system próbuje ukonkretnić znów pierwszą wartość pod zmienną Niebieski:
```
(Zebra=1, Jaguar=2 ,Slimak=3,Czerwony=2, Niebieski=1, Zielony,
Anglik, Japonczyk, Hiszpan)
```
W tym punkcie po raz kolejny naruszony jest warunek mówiący o tym, że wartość zmiennej Slimak musi być mniejsza od wartości zmiennej Niebieski, w związku z czym następują kolejne próby podstawienia:
```
(Zebra=1, Jaguar=2 ,Slimak=3,Czerwony=2, Niebieski=2, Zielony,
Anglik, Japonczyk, Hiszpan)
```

```
(Zebra=1, Jaguar=2 ,Slimak=3,Czerwony=2, Niebieski=3, Zielony,
Anglik, Japonczyk, Hiszpan)
```

Obie te próby zawodzą i następuje kolejne cofnięcie i podstawienie nowej wartości pod zmienną Czerwony:

```
(Zebra=1, Jaguar=2 ,Slimak=3,Czerwony=3, Niebieski, Zielony,
Anglik, Japonczyk, Hiszpan)
```

W tym punkcie nie jest nic naruszone, dlatego następuje po raz kolejny ukonkretnienie zmiennej Niebieski. Za każdym razem jednak po podstawieniu naruszony jest ostatni warunek predykatu „zebra":

```
(Zebra=1, Jaguar=2 ,Slimak=3,Czerwony=3, Niebieski=1, Zielony,
Anglik, Japonczyk, Hiszpan)
(Zebra=1, Jaguar=2 ,Slimak=3,Czerwony=3, Niebieski=2, Zielony,
Anglik, Japonczyk, Hiszpan)
(Zebra=1, Jaguar=2 ,Slimak=3,Czerwony=3, Niebieski=3, Zielony,
Anglik, Japonczyk, Hiszpan)
```

Po tych testach program cofa się próbując znaleźć inne możliwości ukonkretnienia poprzednich zmiennych. Dla zmiennej Czerwony się nie udaje, gdyż nie ma już więcej stałych do podstawienia, podobnie dla zmiennej Slimak. Dopiera dla zmiennej Jaguar możliwe jest podstawienie ostatniej wartości, czyli „3":

```
(Zebra=1, Jaguar=3 ,Slimak,Czerwony, Niebieski, Zielony, Anglik,
Japonczyk, Hiszpan)
```

Ponieważ to podstawienie nie powoduje naruszenia żadnego warunku następuje ukonkretnienie kolejnej zmiennej, czyli Slimak pierwszą możliwą wartością, czyli „1", gdzie zachodzi konflikt, wobec czego system próbuje podstawić kolejną wartość, czyli 2:

```
(Zebra=1, Jaguar=3 ,Slimak=1,Czerwony, Niebieski, Zielony, Anglik,
Japonczyk, Hiszpan)
```

i dla takiego ustawienia ograniczenie nie jest naruszone:

```
(Zebra=1, Jaguar=3 ,Slimak=2,Czerwony, Niebieski, Zielony, Anglik,
Japonczyk, Hiszpan)
```

Następuje więc podstawienie wartości za kolejną zmienną:

```
(Zebra=1, Jaguar=3 ,Slimak=2,Czerwony=1, Niebieski, Zielony,
Anglik, Japonczyk, Hiszpan)
```

ponieważ nic nie jest naruszone ukonkretnia się kolejną zmienną pierwszą dostępną wartością:

```
(Zebra=1, Jaguar=3 ,Slimak=2,Czerwony=1, Niebieski=1, Zielony,
Anglik, Japonczyk, Hiszpan)
```

Występuje konflikt (wartość zmiennej Czerwony jest równa wartości zmiennej Niebieski), wobec czego następuje kolejna próba podstawienia:

```
(Zebra=1, Jaguar=3 ,Slimak=2,Czerwony=1, Niebieski=2, Zielony,
Anglik, Japonczyk, Hiszpan)
```

Gdzie znów jest naruszony ostatni warunek predykatu „zebra" tzn. wartość zmiennej Niebieski nie jest większa od wartości zmiennej Slimak. W związku z tym system podstawia ostatnią dostępną wartość pod zmienną Niebieski i w tym punkcie żaden warunek nie jest naruszony:

```
(Zebra=1, Jaguar=3 ,Slimak=2,Czerwony=1, Niebieski=3, Zielony,
Anglik, Japonczyk, Hiszpan)
```

W związku z tym następuje kontynuacja procesu i system podstawia pierwszą dostępną wartość za zmienną Zielony i tak dalej kontynuuje się podstawianie aż wyczerpią się wszystkie możliwości. Gdy system w pewnym momencie znajdzie rozwiązanie, które spełni wszystkie warunki to je poda a następnie będzie próbował szukać dalej aż do wyczerpania możliwości podstawiania. Algorytm standard backtracking, nawet bez żadnej heurystyki przeszukiwania jest znacznie efektywniejszy od np. przeglądu zupełnego: warto zwrócić uwagę, że podczas kolejnych ukonkretnień i nawrotów system od razu odrzuca bardzo wiele z góry „przegranych" układów podstawień stałych za zmienne np. gdy w jednym z pierwszych kroków system odrzucił podstawienie:

```
(Zebra=1, Jaguar=1 ,Slimak,Czerwony, Niebieski, Zielony, Anglik,
Japonczyk, Hiszpan)
```

to odrzucił również wszystkie przypadki, w których oprócz tych zmiennych ukonkretnione były inne, np:

```
(Zebra=1, Jaguar=1 ,Slimak=2,Czerwony=1, Niebieski=2, Zielony=3,
Anglik=1, Japonczyk=2, Hiszpan=3)
(Zebra=1, Jaguar=1 ,Slimak=2,Czerwony=1, Niebieski=2, Zielony=3,
Anglik=1, Japonczyk=2, Hiszpan=2)
```

...

które to wszystkie przypadki ukonkretnień musiałyby być wygenerowane i przetworzone podczas rozwiązywania zadania za pomocą przeglądu zupełnego. Oczywiście są również algorytmy bardziej efektywne (część z nich będzie wspomniana m.in. w rozdziale poświęconym programowaniu w logice z ograniczeniami) ale język PROLOG nie jest językiem stworzonym do rozwiązywania problemów przeszukiwania a raczej reprezentacji wiedzy i wnioskowania.

2.2.5. Modelowanie procesów decyzyjnych

Logika jest nauką o poprawnym rozumowaniu, język logiki jest w związku z tym doskonałym narzędziem do modelowania procesów rozumowania. Praktyczna implementacja programowania w logice jaką jest język PROLOG ma szereg cech, które ułatwiają modelowanie i symulacje procesów rozumowania. Jedną z podstawowych jego zalet jest eliminacja przepaści

semantycznej między programem a modelem sporządzonym w języku logiki, dzięki czemu stosunkowo prosto można przejść z modelu logicznego do gotowego programu. Poniżej zostanie zaprezentowane kilka przykładowych modeli sytuacji decyzyjnych wymagających wnioskowania. W większości rzeczywistych problemów człowiek rzadko posługuje się wnioskowaniem opartym tylko na dedukcji i regule odrywania. Zazwyczaj okazuje się, że wiedza dotycząca problemu jest zbyt fragmentaryczna i brak jest wprost zdefiniowanych reguł i zasad pozwalających dojść do określonego wniosku, w związku z czym musi posiłkować się abdukcją, indukcją, wnioskowaniem instrumentalnym, z przeciwieństw, z większego na mniejsze itp.

2.2.5.1. Informacje wstępne

Zanim zostaną zademonstrowane przykłady warto przyjrzeć się nieco ogólniej całemu problemowi. Generalnie większość opisanych wyżej przykładów można nazwać modelami jakiegoś procesu rozumowania, ale w tym rozdziale autor proponuje zawęzić problem i spojrzeć na sytuacje decyzyjne jako pewien proces, w którym na podstawie faktów i wiedzy generuje się nowe fakty będące podstawą do podjęcia pewnych decyzji. Znów jest to bardzo ogólna definicja a istotna dla całego modelu jest tu konieczność jawnego zdefiniowania wiedzy dotyczącej problemu oraz wiedzy dotyczącej samego dojścia do rozwiązania, ponieważ wiele specyficznych sytuacji decyzyjnych wymaga specyficznych sposobów wnioskowania. Ze względu na samą naturę języka, formalnie rzecz biorąc, nie ma rozróżnienia w języku na wiedzę dotycząca problemu i wiedzę dotycząca samego dojścia do rozwiązania: tu i tu programista ma do czynienia z takimi samymi regułami a rozróżnienie to ma znaczenie tylko umowne.

2.2.5.2. Przykłady

Bardzo dobrym materiałem źródłowym do przykładów jest wnioskowanie prawnicze. Prawo jest, z jednej strony dziedziną, w której wyroki, decyzje powinny być precyzyjne i jednoznaczne, a z drugiej strony opiera się ona na bardzo często niejednoznacznej ocenie rzeczywistości, wieloznaczności przepisów, trudności w ich interpretacji, konfliktach itp. Wykorzystuje ono bardzo wiele sposobów wnioskowania, które często są zawodne tzn. nie zawsze dają poprawne wnioski, a mechanizmy wnioskowania wykorzystywane w prawie opierają się bardzo mocno na wiedzy zdroworozsądkowej, która jest bardzo trudna (bądź niemożliwa) do pełnej implementacji. Ze względu na poziom skomplikowania i osadzenie w wiedzy zdroworozsądkowej jest to jeden z najtrudniejszych do symulacji procesów rozumowania. Poniższe przykłady dotyczą wykorzystania kilku wybranych mechanizmów wnioskowania i wykorzystywanych w prawie, pozwalają one na zrozumie-

nie i zamodelowanie pewnych mechanizmów wykorzystywanych generalnie podczas rozumowania.

— Wnioskowanie z przeciwieństw:
Częstym problemem wnioskowania prawniczego (i nie tylko prawniczego) jest konieczność podejmowania decyzji dla sytuacji nie unormowanej wprost, czyli takiej, co do której nie zdefiniowano żadnych zasad wprost regulujących jak taki problem rozwiązać. Teoria i praktyka prawa zdefiniowała szereg mechanizmów radzenia sobie z tego rodzaju problemami. Jedną z metod radzenia sobie z tego rodzaju problemami jest wnioskowanie a' contrario. Wnioskowanie z przeciwieństwa zakłada, że jeżeli jakaś norma zakazuje czynu X to dozwolony jest czyn Y będący czynem przeciwnym do X [10]. Chcąc zamodelować tego rodzaju rozumowanie należy w pierwszej kolejności zdefiniować predykaty opisujące stany deontyczne akcji tzn. to, czy dana akcja jest dozwolona, zabroniona, czy zakazana. Przykład dotyczył będzie prostej sytuacji drogowej: samochód stojący na skrzyżowaniu ma możliwość skręcenia w prawo, bądź w lewo, przy czym na tym skrzyżowaniu obowiązuje zakaz skręcania w lewo. Nic nie reguluje kwestii, czy można skręcić w prawo.
Model sytuacji:

```
akcja(skret_w_lewo).
akcja(skret_w_prawo).
zabronione(skret_w_lewo).
```

Niestety z owego modelu nie da się wywnioskować, czy dozwolony jest skręt w prawo. W związku z czym konieczna jest implementacja reguły a'contrario. Aby było to możliwe konieczne jest określenie, które akcje są względem siebie przeciwne. Można to zrealizować definiując dwu (lub więcej) argumentowy predykat „przeciwnosc", w którym oznaczane będą akcje względem siebie przeciwne, np.:

```
przeciwnosc(skret_w_lewo, skret_w_prawo).
```

Mając tego rodzaju wiedzę bardzo łatwo można zaimplementować regułę a'contrario:

```
dozwolone(X):- akcja(X), zabronione(Y), przeciwnosc(X,Y).
```

ponieważ relacja reprezentowana przez predykat „przeciwnosc" jest symetryczna (czyli jeśli X jest przeciwne do Y, to Y jest przeciwne do X) warto dodać drugą definicję reguły a'contrario:

```
dozwolone(X):- akcja(X), zabronione(Y), przeciwnosc(Y,X).
```

Taka definicja może łatwo prowadzić do powstania konfliktu: gdyby zabroniony był również skręt w prawo, to system wywnioskowałby, że skręt w lewo (i w prawo) jest jednocześnie dozwolony i zabroniony (zabroniony z definicji wprost a dozwolony z reguły a'contrario). Jest to oczywisty absurd, dlatego konieczne jest dodanie klauzuli mówiącej, że z reguły a'contrario dozwolone mogą być tylko te akcje, które nie zostały wcze-

śniej zabronione:

```
dozwolone(X):- akcja(X), zabronione(Y), przeciwnosc(X,Y), not
zabronione(X).
dozwolone(X):- akcja(X), zabronione(Y), przeciwnosc(Y,X), not
zabronione(X).
```

To co zostało (w bardzo uproszczony sposób) zamodelowane powyżej obrazuje właśnie włączenie pewnego fragmentu wiedzy zdroworozsądkowej (predykat „przeciwnosc" – opisuje oczywistą zazwyczaj dla człowieka relację przeciwności) i zdroworozsądkowego wnioskowania (a'contrario) w praktyce. Warto zauważyć, że tego rodzaj wnioskowanie jest zawodne (nie zawsze musi prowadzić do prawdziwych wniosków) oraz ma charakter podważalny i niemonotoniczny (coś raz uznane za dozwolone może przy dodaniu nowej wiedzy zostać uznane za zabronione). Wnioskowanie niemonotoniczne zostanie omówione w dalszych rozdziałach książki.

— Wnioskowanie abdukcyjne:
Wnioskowanie abdukcyjne ma charakter przeciwny do klasycznego wnioskowania dedukcyjnego. O ile podczas klasycznego wnioskowania z wykorzystaniem reguły odrywania prawdziwość wniosku reguły jest potwierdzona, gdy prawdziwe są jej przesłanki, czyli:
Prawdziwe jest A i
$A \rightarrow B$
W związku z czym prawdziwe jest B, o tyle podczas wnioskowania abdukcyjnego, gdy:
Prawdziwe jest B i
$A \rightarrow B$
W związku z czym zakłada się że prawdziwe jest A. Wnioskowanie abdukcyjne ma charakter zawodny, czyli nie zawsze jest spełnione, jest podobnie jak wnioskowanie z przeciwieństw niemonotoniczne i podważalne (pojawienie się nowego faktu przeczącego prawdziwości A obala je). Wnioskowanie abdukcyjne można traktować jako narzędzie do wyjaśniania rzeczy już wiadomych. Wnioskowanie abdukcyjne można zilustrować przykładem próby wywnioskowania, czy samochód „ford", który właśnie przyjechał, przeszedł badanie techniczne. Ponieważ istnieje reguła prawna, która mówi, że samochód bez badania technicznego nie będzie dopuszczony do ruchu i skoro samochód ten był dopuszczony do ruchu to korzystając z wnioskowania abdukcyjnego można założyć, że samochód ów przeszedł badanie techniczne.

Zanim powstanie model powyższego wnioskowania konieczne jest zwrócenie uwagi na pewien bardzo istotny szczegół: warunkiem zadziałania wnioskowania abdukcyjnego jest sprawdzenie, czy istnieje reguła, która

w swoim wniosku ma zadeklarowany zadeklarowany już fakt. Sprawdzenie to wymaga użycia kwantyfikatora egzystencjalnego na regułę („czy istnieje taka reguła, w której wnioskiem jest, że samochód jest dopuszczony do ruchu"). Jest to przykład logiki wyższego rzędu, w której możliwe jest wykorzystanie kwantyfikatorów na funkcje bądź reguły. Ponieważ w logice pierwszego rzędu ani w programowaniu logicznym tego rodzaju działania nie są możliwe, to konieczne będzie wykorzystanie pewnej „sztuczki" tak aby obejść owo ograniczenie. „Chwyt" ów polega na tym, że przesuwa się niejako regułę do poziomu argumentu, na podstawie której wykonuje się dopiero proces wnioskowania. Definiuje się predykat „regula", dla którego pierwszy argument będzie wnioskiem reguły, drugi argument (i kolejne argumenty, bo można dać kilka definicji predykatu z różną liczbą argumentów) jest warunkiem do spełnienia dla danej reguły. Dla najprostszego przykładu wnioskowania dedukcyjnego może to wyglądać tak:

```
samochód(ford).
regula(dopuszczony_do_ruchu,badanie_techniczne).
zachodzi(dopuszczony_do_ruchu, ford).
zachodzi(Wniosek,Obiekt):-regula(Wniosek,Warunek),
zachodzi(Warunek,Obiekt).
```

Program działa w ten sposób, że dwuargumentowy predykat „zachodzi" opisuje, że występuje jakaś cecha, bądź właściwość wymieniona w pierwszym argumencie, odnosząca się do drugiego argumentu (w tym wypadku samochód ford jest dopuszczony do ruchu). Na podstawie reguły zdefiniowanej w predykacie „regula" określa się, czy zachodzi dana właściwość względem danego obiektu. Zadając zapytanie do systemu:

```
zachodzi(Wniosek,Obiekt).
```

otrzyma się odpowiedź:

```
Wniosek = badanie_techniczne,
Obiekt = ford ;
Wniosek = dopuszczonydo_ruchu,
Obiekt = ford
```

Co można interpretować tak, że zachodzi fakt ,iż samochód ford ma badanie techniczne i zachodzi fakt, iż samochód ów będzie dopuszczony do ruchu. Jest to oczywiście bardzo uproszczony model wnioskowania, w którym, między innymi, warunek i wniosek dotyczą tego samego argumentu (samochód ford), co przecież nie zawsze musi zachodzić, bez większych jednak trudności można rozbudować system dając szersze możliwości modelowania reguł.

Wnioskowanie abdukcyjne w przeciwieństwie do wnioskowania dedukcyjnego ma charakter zawodny, w związku z czym nieco dyskusyjne byłoby używanie tego samego predykatu „zachodzi" jaki wykorzystuje się w

przypadku wnioskowania dedukcyjnego. Proponuje się wobec tego wykorzystanie tu predykatu „zachodzi_podważalnie", który reprezentowałby takie występowanie danej cechy, które może zostać podważone przez inne mechanizmy wnioskowania. Definicja wnioskowania abdukcyjnego dla przykładu kolejnego samochodu może wyglądać tak:

```
zachodzi(badanie_techniczne, audi).
zachodzi_podwazalnie(Warunek,Obiekt):- regula(Wniosek, Warunek),
zachodzi(Wniosek,Obiekt).
```

Chcąc dowiedzieć się, czy w sposób podważalny zachodzi jakaś relacja należy zadać pytanie:

```
zachodzi_podwazalnie(X,Y).
```

Odpowiedź systemu będzie wtedy taka:

```
X = badanie_techniczne,
Y = ford
```

Co można zinterpretować tak, że mimo tego, że nie da się określić jednoznacznie, czy samochód ford ma badanie techniczne, to korzystając z rozumowania abdukcyjnego można stwierdzić, iż są pewne przesłanki potwierdzające, że fakt ten zachodzi. Powyższa reprezentacja podważalności ma jednak pewną wadę: nie pozwala ona na dokonywanie zapytań ogólnych, bez doprecyzowywania, czy wywnioskowany, bądź zadeklarowany fakt jest podważalny, czy też nie. Najprościej problem ten można rozwiązać przenosząc informację o tym, czy fakt jest podważalny, czy nie, do argumentu predykatu „zachodzi". Wtedy cały program powinien wyglądać tak:

Listing 2.11. Wnioskowanie abdukcyjne

```
1 samochod(ford).
  regula(dopuszczony_do_ruchu,badanie_techniczne).
3 zachodzi(dopuszczony_do_ruchu, ford, niepodwazalnie).
  zachodzi(Wniosek, Obiekt, Podwazalnosc):-regula(Wniosek,
5     Warunek), zachodzi(Warunek,Obiekt, Podwazalnosc).
  zachodzi(badanie_techniczne, audi, niepodwazalnie).
7 zachodzi(Warunek, Obiekt, podwazalnie):- regula(Wniosek,
      Warunek), zachodzi(Wniosek,Obiekt, _).
```

Predykat „zachodzi" ma w tej sytuacji aż trzy argumenty i możliwe jest swobodne dokonywanie zapytań o to, czy dany fakt zachodzi podważalnie, niepodważalnie, bądź bez doprecyzowywania tego parametru.

— Wnioskowanie z mniejszego na większe (z większego na mniejsze). Wnioskowanie to działa wg schematu: jeżeli A to tym bardziej B. Występuje ono w kilku odmianach m.in. skoro zabronione jest coś mniejszego to tym bardziej zabronione jest coś większego (jeżeli zabronione korzystanie ze

ściąg na egzaminie to tym bardziej zabronione jest odpisywanie pracy od kolegi). Chcąc zamodelować tego rodzaju proces wnioskowania konieczne jest najpierw zdefiniowanie predykatów opisujących dostępne akcje oraz pojęcia deontyczne (podobnie jak dla wnioskowania a'contrario):

```
akcja(korzystanie_ze_sciag).
akcja(odpisywanie_od_kolegi).
zabronione(korzystanie_ze_sciag).
```

Następnie warto zwrócić uwagę na to, że podczas wnioskowania z mniejszego na większe kluczowym czynnikiem jest relacja większy-mniejszy między analizowanymi akcjami. Ze względu na to, że trudno jest z góry zdefiniować jaki charakter ta relacja może mieć oraz to, że takich relacji może być więcej (np. Flip jest wyższy od Flapa, ale Flap jest cięższy od Flipa) to bardzo trudno byłoby zdefiniować jakąś uniwersalną i formalną metodę oceny tej relacji. Niestety, póki co, jedynym sposobem jej definicji jest deklaracja wprost przez programistę, np.:

```
mniejsze(korzystanie_ze_sciag, odpisywanie_od_kolegi).
```

Oczywiście wciąż nie jest pewne czy dana relacja mniejszy-większy jest adekwatna do wysnucia danego wniosku i ze względu na jej ściśle merytoryczny charakter nikt poza końcowym użytkownikiem nie będzie wiedział, czy ta relacja rzeczywiście będzie pasować do określonego wnioskowania, w związku z czym nie jest możliwe w pełni automatyczna realizacja takiego wnioskowania i zawsze człowiek na końcu będzie musiał zweryfikować dobór rodzaju relacji mniejszy-większy między konkretnymi akcjami. Można jednak potraktować tego rodzaju model wnioskowania jako narzędzie doradzające sugerujące pewien kierunek rozumowania. Samo wnioskowanie można zdefiniować za pomocą predykatu:

```
zabronione_podwazalnie(AkcjaW):- akcja(AkcjaW), akcja(AkcjaM),
zabronione(AkcjaM), mniejsze(AkcjaM, AkcjaW).
```

Wnioskowanie z mniejszego na większe jest wnioskowaniem podważalnym, czyli może zostać podważone przez inny argument, w związku z czym wniosek również będzie podważalny. Do opisu stanu deontycznego akcji wykorzystano, podobnie jak w przypadku wnioskowania abdukcyjnego predykat z dopiskiem podważalny, przez co możliwe jest jego rozróżnienie od wyniku wnioskowania i deklaracji niepodważalnych. Taka reprezentacja ma, podobnie jak w poprzednim przykładzie, jednak pewne wady wynikające z różnych nazw predykatów odnoszących się do jednego stanu deontycznego. W takiej sytuacji najlepszym rozwiązaniem jest przesunięcie informacji o podważalności danego wniosku do argumentu oceny deontycznej, czyli:

Listing 2.12. Wnioskowanie większy-mniejszy

```
akcja(korzystanie_ze_sciag).
akcja(odpisywanie_od_kolegi).
zabronione(korzystanie_ze_sciag, niepodwazalnie).
mniejsze(korzystanie_ze_sciag, odpisywanie_od_kolegi).
zabronione(AkcjaW, podwazalnie):- akcja(AkcjaW),
    akcja(AkcjaM), zabronione(AkcjaM, _),
    mniejsze(AkcjaM, AkcjaW).
```

W takim rozwiązaniu możliwe jest dokonanie zapytania o to co jest zabronione podważalnie:
`zabronione(X,podwazalnie).`
Niepodważalnie:
`zabronione(X,niepodwazalnie).`
Bądź bez wyszczególniania jaki jest charakter zakazu:
`zabronione(X,_).`

— Wnioskowanie instrumentalne. Wnioskowanie instrumentalne mówi, że skoro prawdą jest A, to B, które jest konieczne do tego aby A było prawdą, również musi być prawdą. Definicję można poprzeć przykładem:
Aby ktoś zdał egzamin koniecznie musi się nauczyć.
Zosia zdała egzamin, stąd wniosek, że się nauczyła
Mechanizm ten jest w swej istocie zbliżony do wnioskowania abdukcyjnego, różnica leży w określeniu „konieczny": Nie może istnieć sposób udowodnienia wniosku (Zosia zdała egzamin) dla którego warunek (Zosia się nauczyła) będzie fałszem. W klasycznym wnioskowaniu może być więcej reguł o tym samym wniosku:
`regula(zdal_egzamin, nauczyl_sie).`
`regula(zdal_egzamin, sciagnal).`
Z powyższego przykładu widać, że mógł zajść mało prawdopodobny, ale niewykluczony przypadek, że student (np. Zosia) nie nauczył się na egzamin, a ściągnął go od kolegi. W takiej sytuacji nie można powiedzieć, że nauczenie się jest warunkiem koniecznym do zdania egzaminu (choć oczywiście zazwyczaj jest warunkiem wystarczającym). Gdyby jednak założyć, że w analizowanych regułach będą po dwa warunki w takiej postaci:
`regula(zdal_egzamin, nauczyl_sie, przyszedl_na_egzamin).`
`regula(zdal_egzamin, sciagnal, przyszedl_na_egzamin).`
Czyli do zdania egzaminu, oprócz nauki (bądź ściągnięcia), wymagana jest również obecność na tym egzaminie. Można tu zauważyć, że obecność owa jest dla analizowanego przypadku warunkiem koniecznym. Problem oceny, czy dany warunek jest rzeczywiście konieczny jest znów za-

leżny od merytoryki i trudno byłoby wskazać jakąś uniwersalną formalną metodę badania „stopnia konieczności". Można ją intuicyjnie wywnioskować z tego, że warunek konieczny pojawia się we wszystkich regułach, w których wnioskiem jest „zdal_egzamin". Jest to jednak bardzo zawodny sposób. Najprościej po prostu zdefiniować predykat „konieczne", który wyrażałby ową relację między stałymi. Opisaną wyżej sytuacji można w związku z tym wyrazić tak:

Listing 2.13. Wnioskowanie instrumentalne

```
regula(zdal_egzamin, nauczyl_sie, przyszedl_na_egzamin).
regula(zdal_egzamin, sciagnal, przyszedl_na_egzamin).
zachodzi(nauczona, zosia, niepodwazalnie).
zachodzi(zdal_egzamin, zosia, niepodwazalnie).
zachodzi(Wn, Obiekt, Podw):-regula(Wn, War1, War2),
    zachodzi(War1, Obiekt, Podw),
    zachodzi(War2, Obiekt, Podw).
zachodzi(Wn, Obiekt, podwazalnie):-regula(Wn, War1, War2),
    zachodzi(War1, Obiekt, Podw1),
    zachodzi(War2, Obiekt, Podw2), Podw1=/=Podw2.
zachodzi(War1, Obiekt, podwazalnie):-regula(Wn, War1, War2),
    zachodzi(Wn, Obiekt, _), zachodzi(War2, Obiekt, _),
    konieczne(War1, Wn).
zachodzi(War2, Obiekt, podwazalnie):-regula(Wn, War1, War2),
    zachodzi(Wn, Obiekt, _), zachodzi(War1, Obiekt, _),
    konieczne(War2, Wn).
```

Przykład jest bardzo prosty a ewentualne wątpliwości czytelnika mogą budzić dwie reguły dotyczące wnioskowania klasycznego: w jednej wniosek jest podważalny jeśli podważalne są oba warunki i niepodważalny jeśli niepodważalne są oba warunki, w drugim zaś, gdy jeden z warunków jest podważalny, a drugi nie, to wniosek zawsze jest podważalny (zasada łańcucha: jest tak mocny jak mocne jest jego najsłabsze ogniwo).

Podsumowując: powyższe modele mechanizmów rozumowania są, oczywiście, tylko bardzo szkicowymi próbami ich implementacji, aby zachować czytelność i prostotę nie uwzględniono w nich wielu utrudnień i pułapek, które czyhają na programistę chcącego symulować ludzkie rozumowanie. Uważny czytelnik niewątpliwie znajdzie mnóstwo takich przykładów, które wykroczą poza zdefiniowane tu modele. Znakomitym ćwiczeniem byłaby próba modyfikacji powyższych programów tak, aby potrafiły prawidłowo wnioskować dla bardziej skomplikowanych problemów.

Inne mechanizmy rozumowania takie jak np. analogia, bądź indukcja są ze względu na ich osadzenie w wiedzy zdroworozsądkowej, a także ich własną specyfikę bardzo trudne do implementacji, także w języku PROLOG. Mechanizm zbliżony do analogii jest najczęściej stosowany we wnioskowaniu na

bazie przykładów (Case Based Reasoning), gdzie najczęściej wykorzystywany jest algorytm k-najbliższych sąsiadów do szukania podobieństw między analizowanymi przypadkami. Wnioskowanie indukcyjne opiera się na wysnuwaniu ogólnych twierdzeń na podstawie konkretnych przykładów: jeśli wiadomo, że Zosia jest studentem i zdała egzamin i Janek jest studentem i zdał egzamin, to wiedząc, że Maciek jest studentem też można założyć, że zdał egzamin. Jest to oczywiście wnioskowanie zawodne, ale często stosowane. Więcej na jego temat będzie w rozdziale poświęconym uczeniu się.

2.3. Uwagi końcowe

Powyższe wprowadzenie do programowania w logice nie ma na celu, jak już wcześniej zostało napisane, być kompletnym kursem języka PROLOG, tym bardziej, że taka pozycja jest dostępna w Polsce [7], jest też mnóstwo źródeł internetowych i kursów on-line, które pozwalają na swobodną samodzielną naukę. Również producenci środowisk programistycznych dla języka PROLOG oferują bardzo wiele materiałów (np. www.swi-prolog.org), z których można i trzeba korzystać. Język PROLOG nie jest językiem uniwersalnym i o ile znakomicie sprawdza się przy modelowaniu wnioskowania i zależności logicznych o tyle w całym szeregu zadań innego rodzaju (grafika, obliczenia itp.) jest znacznie mniej użyteczny niż „klasyczne" proceduralne i obiektowe języki programowania. W związku z tym w dzisiejszych czasach bardzo rzadko wykorzystuje się go jako narzędzie do tworzenia całych aplikacji. Znacznie częściej jest wykorzystywany do budowy specjalizowanych modułów włączanych do większych programów bądź w zastosowaniach akademickich do modelowania zależności logicznych, automatycznego dowodzenia twierdzeń przetwarzania języka naturalnego itp.

Względna łatwość z jaką modeluje się problemy przeszukiwania stała się jedną z inspiracji rozwoju języka. Pod koniec lat osiemdziesiątych opracowano nową klasę języków programowania, które bazując na składni i paradygmacie języka PROLOG zostały wzbogacone o ulepszone mechanizmy przeszukiwania i mechanizmy optymalizacji. Ze względu na to, iż powstały one w celu wykorzystania do rozwiązywania problemów optymalizacyjnych a nie modelowania wiedzy i wnioskowania to języki klasy CLP (Constraint Logic Programming), bo o nich mowa, zostaną dokładniej opisane w dalszych rozdziałach.

Rozdział 3

Systemy ekspertowe

Systemy ekspertowe stały się pierwszymi narzędziami utożsamianymi z terminem „sztuczna inteligencja", które znalazły zastosowanie w praktyce. Pierwsze próby realizacji takich narzędzi rozpoczęły się w czasach będących z dzisiejszego punktu widzenia prehistorią informatyki. W połowie lat sześćdziesiątych powstał system Dendral, który był narzędziem wspomagającym analizę struktury molekularnej chemicznych związków organicznych na podstawie analizy widm spektroskopowych. System ten zawierał w sobie moduły proceduralne i deklaratywne, został napisany w języku LISP. Stał się on inspiracją i podstawą do rozwoju dla innych systemów ekspertowych takich jak system Prospector, czy MYCIN. Zadaniem systemu Prospector było wspomaganie pracy geologów jego budowa rozpoczęła się w latach 70-tych i trwała aż do roku 1983. Bardzo spektakularnym sukcesem okazało się wykrycie za pomocą systemu dużych złóż molibdenu. Najsławniejszym jednak systemem okazał się opracowany pod koniec lat 70-tych system MYCIN badający choroby bakteryjne krwi. System wykazywał się dużą skutecznością (porównywalną ponoć z poziomem lekarza po studiach) i szybkością. Był wykorzystywany w praktyce do szkolenia studentów. System ten był jednym z pierwszych, które wykorzystywały reprezentację wiedzy niepewnej.

Wokół systemów ekspertowych narosło sporo mitów, które powielane przez wiele źródeł funkcjonują do dziś zafałszowując ich faktyczne możliwości. Owe mity z jednej strony są zbyt „hurra-optymistyczno" entuzjastyczne a z drugiej zbyt deprecjonujące ich faktyczny potencjał. Do dziś w niektórych pozycjach autorzy traktują systemy ekspertowe jako znakomite narzędzie nadające się dla każdego i wspomagające rozwiązywanie niemalże każdego problemu, czasem przypisując im możliwości, których nigdy miały (m.in. rozwiązywanie bliżej nie sprecyzowanych „nieustrukturalizowanych" problemów). Z drugiej strony gdzieniegdzie można znaleźć pozycje traktują systemy ekspertowe jako przestarzałą technologię, z której nikt poza pracownikami uniwersytetów nie korzysta. Oba te poglądy są dość dalekie od prawdy, co postara się poniższy rozdział wyjaśnić.
Wyróżnione innowacyjne rozwiązania, wykorzystujące techniki sztucznej inteligencji jakie zostały przedstawione na konferencji „The two faces of AI '97" organizowanej przez AAAI w zdecydowanej większości były zastosowanymi do wspomagania podejmowania różnych decyzji systemami ekspertowymi [3].
Nazwa systemy ekspertowe pochodzi się od słowa ekspert, oznaczającego człowieka dysponującego specjalistyczną wiedzą w pewnej dziedzinie i umiejącego i potrafiącego wykorzystać do rozwiązywania problemów z tej dziedziny.

Systemy ekspertowe przez różnych autorów były w różny sposób defi-

niowane: wg Włodzisława Ducha [8] system ekspertowy jest to inteligentny program komputerowy wykorzystujący procedury wnioskowania do rozwiązywania tych problemów, które są na tyle trudne, że normalnie wymagają znaczącej ekspertyzy specjalistów. J. Mulawka [14] definiuje je jako programy komputerowe przeznaczone do rozwiązywania specjalistycznych problemów wymagających profesjonalnej ekspertyzy. Dodaje też, że charakteryzują się odseparowaniem wiedzy dziedzinowej dotyczącej analizowanego problemu od reszty systemu. Buchannan i in. [5] definiują system ekspertowy jako: program komputerowy który stosuje modele wiedzy i procedury wnioskowania w celu rozwiązania problemów, podobnie [16] definiuje je jako: „programy do rozwiązywania problemów zalecanych ekspertom, charakteryzujący się strukturą funkcjonalną, które podstawowymi elementami są:

1. Baza wiedzy zawierająca wiedzę potrzebną do rozwiązania określonego problemu (..)
2. System wnioskujący wyznaczający fakty wynikające z bazy wiedzy i zbioru faktów początkowych"

Część definicji zwraca uwagę na przeznaczenie systemów ekspertowych, inne zaś koncentrują się na ich budowie i cechach charakterystycznych odróżniających je od klasycznych programów komputerowych. Ostatnia definicja koncentruje się na głównej charakterystyczne cesze systemów ekspertowych tj. rozdziale wiedzy dziedzinowej od procedur wnioskowania. Jeden z twórców systemu DENDRAL, E. Fingenbaum stwierdził, że „Paradygmat systemów ekspertowych pochodzi z wiedzy jaką one posiadają a nie z formalizmów i schematów wnioskowania jakie stosują. Wiedza eksperta jest kluczem systemu, podczas gdy reprezentacja wiedzy i schematy wnioskowania dostarczają tylko mechanizmów jej użycia.". Duży nacisk jaki został tu położony na wiedzę zawartą w bazie podkreśla jej kluczowy wpływ na jakość wyciągniętej konkluzji.

Różne sposoby reprezentacji wiedzy pozwalają w różny sposób przedstawić rzeczywistość danego problemu. Niektóre sposoby pozwalają precyzyjniej opisać istniejące zależności, niektóre zaś umożliwiają prostsze i bardziej zwięzłe wyrażenie wiedzy eksperta. Niektórych zależności nie da się przedstawić wykorzystując pewne sposoby reprezentacji wiedzy. Mulawka w [14] podaje, że reprezentacja wiedzy stanowi jeden z podstawowych problemów, który nie został jeszcze rozwiązany. Wynika to m.in. ze skomplikowanej i niejednorodnej natury opisywanej rzeczywistości. Opisem sposobu „reprezentacji wiedzy" w umyśle ludzkim zajmuje się psychologia poznawcza i często na jej bazie powstają różne sposoby wyrażania wiedzy w systemach ekspertowych. Mulawka w [14] definiuje wiedzę jako: „zbiór wiadomości z określonej dziedziny, wszelkie zobiektywizowane i utrwalone formy kultu-

ry umysłowej i świadomości społecznej powstałe w wyniku kumulowania się doświadczeń i uczenia się." Dodaje też, że: „wiedza jest symbolicznym opisem otaczającego nas świata rzeczywistego charakteryzującym aksjomatyczne i empiryczne relacje, zawierającym procedury, które manipulują tymi relacjami." W związku z tym wiedza jako tak zdefiniowane pojęcie składa się z faktów, relacji i procedur. W dalszej kolejności konieczne jest zdefiniowanie czym jest reprezentacja wiedzy: „Reprezentacja wiedzy jest procesem formalizacji pozyskanej od eksperta wiedzy i wyrażenia jej w odpowiedniej, symbolicznej formie, którą będzie można następnie interpretować" [14].

Jedną z cech charakterystycznych systemów ekspertowych jest to, że koncentrują się one bardziej na przedstawieniu wiedzy symbolicznej niż numerycznej, wynika to m.in. z tego iż umysł ludzki „woli" pracę na różnego rodzaju pojęciach i relacjach niż na liczbach. Pomimo tego we współczesnych systemach ze względów praktycznych (do większości zastosowań praktycznych w biznesie, technice itp. przetwarzanie liczb, chociażby najprostsze, jest niezbędne) również wprowadzono moduły przetwarzające wiedzę o charakterze numerycznym.
Literatura [14] podaje, że istnieją generalnie dwa podstawowe sposoby reprezentacji wiedzy:

— proceduralna polegająca na zdefiniowaniu szeregu procedur, których działanie przedstawia wiedzę o dziedzinie,
— deklaratywna polegająca na określeniu zbioru specyficznych dla danej dziedziny faktów oraz relacji między nimi.

Generalnie w systemach ekspertowych stosuje deklaratywny sposób reprezentacji wiedzy. Proceduralny sposób wyrażania wiedzy jest charakterystyczny dla klasycznych programów komputerowych, gdzie w algorytmie programu jest przedstawiony sposób rozwiązania problemu. W systemach ekspertowych nie podaje się sposobu rozwiązania problemu, zadaniem systemu jest właśnie rozwiązać problem, bazując na zawartej w bazie wiedzy, bez podania drogi jego rozwiązania. Niektórzy autorzy [15] wręcz przeciwstawiają systemy ekspertowe jako programy z deklaratywnym sposobem reprezentacji wiedzy klasycznym, programom jako narzędziom wykorzystującym proceduralny sposób reprezentacji wiedzy. Pomimo tego w niektórych systemach stosowanych w praktyce wykorzystuje się oba sposoby równolegle.

Przyglądając się powyższym definicjom, wyróżniającą cechą systemów ekspertowych jest ich specyficzna budowa tzn. wyraźne rozdzielenie wiedzy od wnioskowania. Czytelnicy, którzy pamiętają rozdział poświęcony programowaniu w logice i językowi PROLOG, podział ten, mimo że wprost nie był

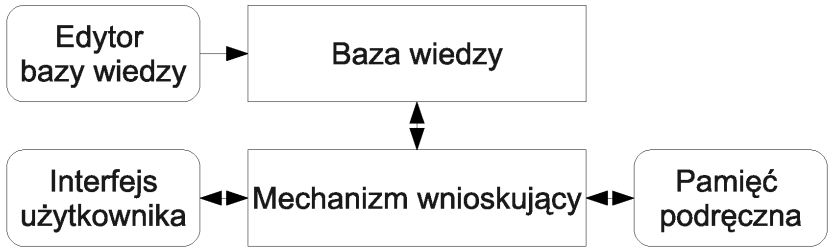
tam wspomniany, nie jest obcy: tam również mechanizm wnioskowania był oddzielony od programu, który można traktować jako bazę wiedzy. Taka budowa wiąże się z szeregiem cech wyróżniających tego rodzaju narzędzie:

— Wiedza jest niezależna od procedur wnioskowania. Wiele badań potwierdza, że eksperci przy podejmowaniu decyzji kierują się przede wszystkim swoją wiedzą, w mniejszym zaś stopniu zaawansowanym wnioskowaniem,
— Łatwość przyrostowej rozbudowy bazy wiedzy: skoro mechanizm wnioskowania i algorytmy są rozdzielone od wiedzy to bez trudu można wiedzę dodać oraz...
— Modyfikowalność. Systemy ekspertowe pozwalają na łatwą modyfikację wiedzy, ma to kluczowe znaczenie dla zastosowań praktycznych, gdzie wiedza szybko się starzeje i staje się nieaktualna. W takim wypadku bez ingerencji w algorytm można bez większych trudności uaktualnić bazę wiedzy,
— Można tworzyć systemy z pustymi bazami wiedzy, do których użytkownik sam będzie mógł wprowadzać wiedzę dotyczącą interesującej go dziedziny. Takie systemy z pustą bazą wiedzy nazywane są szkieletowymi bądź skorupowymi (ang. expert system shell),
— Sposób zakodowania wiedzy w bazie jest zazwyczaj stosunkowo prosty i czytelny nawet dla osoby bez przygotowania informatycznego (szczególnie w porównaniu z programami komputerowymi), dlatego budową i aktualizacją baz wiedzy mogą zajmować się nie tylko informatycy ale także tzw. eksperci dziedzinowi.

Dwie główne części systemu ekspertowego to baza wiedzy i mechanizm wnioskowania. Zazwyczaj podczas wnioskowania mechanizm wykorzystuję podręczną pamięć (zwaną czasem podręczną bazą danych) w której przechowywane są fakty i cząstkowe wyniki wnioskowania. Oprócz tych głównych części większość systemów ekspertowych ma jakiś interfejs użytkownika, edytor bazy wiedzy, czasem jeszcze są moduły ekstrakcji wiedzy pozwalające na generację wiedzy z przykładów, moduły wykorzystujące inne metody sztucznej inteligencji jak np. sieci neuronowe, algorytmy genetyczne i mechanizmy pozwalające na integrację tych wszystkich narzędzi Poniższy rysunek prezentuje ogólny schemat budowy systemu ekspertowego.

3.1. Baza wiedzy

Kluczowym elementem systemu ekspertowego jest jego baza wiedzy. Jeśli mechanizm wnioskowania nie zawiera błędów to jakość ekspertyz wydawanych przez system jest zależna od jakości wiedzy w systemie, dlatego sposób w jaki jest sformalizowana wiedza w systemie ma kluczowe znaczenie dla ja-



Rysunek 3.1. Schemat budowy systemu ekspertowego.

kości jego działania.
Sposoby reprezentacji wiedzy w systemach ekspertowych generalnie w literaturze dzieli się na trzy podstawowe grupy [13], [11], [9]:
— Za pomocą ram,
— Za pomocą sieci semantycznych,
— Za pomocą reguł.
Inni autorzy czasami uszczegóławiają tą listę dodając do niej sposoby przedstawiania faktów w bazie wiedzy (za pomocą logiki dwuwartościowej bazującej na dychotomii prawda–fałsz, logiki wielowartościowej, wykorzystującej strukturę „Obiekt, Atrybut, Wartość", czy logiki rozmytej). Czasem też autorzy do tej listy dołączają np. sieci neuronowe [9], czy algorytmy genetyczne, które są osobnymi dyscyplinami sztucznej inteligencji i mimo tego, że bywają z nimi łączone w systemy hybrydowe, mają zupełnie odmienną filozofię działania.
Wymienione wyżej trzy podstawowe sposoby reprezentacji wiedzy mają już nieco „historyczny" charakter. Na przestrzeni lat część z nich wyszła z użycia, część zaś rozwinęła się i wyewoluowała w zupełnie odmienną od początkowych założeń formę. Oczywiście każdy z tych sposobów bywa implementowany w różny sposób, bywa że sposoby te są ze sobą łączone.

Reprezentacja wiedzy za pomocą ram opiera się wynikach prac psychologii poznawczej, na sposobie rozwiązywania problemów przez ludzi [9], [14]. Wiedza i sposoby rozwiązywania problemów bazują na pewnym modelu struktury wiedzy, zawierające stereotypowe sytuacje. Na tej idei zostało zdefiniowane pojęcie ramy opisującej tą strukturę. Rama ma swą nazwę oraz szczeliny (sloty) również posiadające swe nazwy. Każda szczelina opisuje pewną własność, jest zbiorem możliwych wartości [9], [14]. Konkretna szczelina również może być ramą. Ramy mogą mieć charakter ogólny i mieć

konkretne przypadki, z określonymi wartościami poszczególnych cech. Ramy są ułożone w sposób hierarchiczny, obowiązuje tu zasada dziedziczenia własności. Można je podzielić na ramy „konceptualne" mające charakter ogólny i będące swego rodzaju opisem świata i konkretne przypadki obiektów występujących w tym świecie.

Idea tego sposobu reprezentacji wiedzy opiera się na założeniu, że konkretne przypadki są odzwierciedleniem pewnego zdefiniowanego modelu. Możliwe jest również zdefiniowanie wyjątków od określonego modelu [13]. Ogólnie zdefiniowane ramy określane są jako klasy, ich uszczegółowienie to podklasy, na najniższym poziomie występują określone konkretne przypadki. Na każdym poziomie mają one swoje szczeliny z własnościami, które są dziedziczone przez niższe klasy aż do konkretnych przypadków.

Ramy są mechanizmem pozwalającym na wyrażenie pewnej struktury i hierarchii rzeczywistości. Były one często wykorzystywane do reprezentacji tzw. wiedzy zdroworozsądkowej czyli wiedzy ogólnej dotyczącej jakiegoś fragmentu rzeczywistości, hierarchii pojęć bytów i ich relacji. Reprezentację wiedzy zbliżoną do ram wykorzystuje słynny system tworzony w ramach projektu CYC, który jest próbą wyrażenia w komputerze wiedzy zdroworozsądkowej jaką posługuje się człowiek (http://www.cyc.com/). Podstawową słabością ram jest brak osadzenia w formalizmie matematycznym a także stosunkowo niewielkie możliwości reprezentacji różnego rodzaju zależności. Były one tak naprawdę pierwszą próbą implementacji ontologii w systemie komputerowym (choć wtedy ten termin jeszcze nie był zbyt popularny w informatyce). Ramy w czystej postaci nigdy nie były bardzo popularnym sposobem reprezentacji wiedzy i w dzisiejszych czasach praktycznie wyszły z użycia zastąpione przez ontologie (ontologia to formalny opis dziedziny wiedzy zawierający zbiór pojęć i relacji między nimi, więcej na ich temat będzie w dalszej części rozdziału) wyrażone za pomocą logiki opisowej, które niejako kontynuują ten kierunek sposobu reprezentacji wiedzy. Ontologie wyrażone za pomocą logiki opisowej ostatnimi czasy święcą triumfy w internecie będąc narzędziem do realizacji „semantycznego internetu" (więcej o ontologiach i kryjących się za nimi formalizmach będzie w dalszej części książki).
Niektóre systemy ekspertowe oprócz ram wykorzystują reguły dodatkowo opisujące zależności między ramami.

Idea reprezentacji wiedzy za pomocą sieci semantycznych opiera się na deklaracji pewnych obiektów będących pojęciami z danej dziedziny oraz zbioru krawędzi łączących te obiekty symbolizujących relacje między nimi [13]. Sposób ten został wprowadzony przez Quiliana [14]. Sieć semantyczna

jest swego rodzaju grafem skierowanym. Między węzłami grafu zachodzą relacje, których może być wiele różnych rodzajów, mogą one zachodzić w jedną bądź dwie strony. Reprezentacja wiedzy za pomocą sieci semantycznych w swych założeniach została stworzona do systemów zajmujących się analizą i rozumieniem języka naturalnego [14]. Sieci semantycznie w „czystej postaci" generalnie wyszły z użycia, gdyż podobnie jak w przypadku ram logika deskrypcyjna okazała się znacznie silniejszym, bardziej sformalizowanym i intuicyjnym narzędziem do reprezentacji takiego rodzaju wiedzy (więcej na temat logiki deskrypcyjnej w dalszej części książki).

3.1.1. Reguły

Najpopularniejszym sposobem reprezentacji wiedzy wykorzystywanym w systemach ekspertowych jest reprezentacja wiedzy za pomocą reguł. Oparty jest on na wykorzystaniu reguł typu:

wniosek **jeżeli** lista warunków

przy czym zazwyczaj warunki połączone są spójnikiem logicznym „i" (czyli „and"). Niektóre systemy pozwalają na wykorzystanie także spójnika logicznego „lub" (ang. „or") jednakże zmniejsza to czytelność bazy wiedzy i nie zwiększa możliwości systemu, gdyż bardzo łatwo można go zastąpić poprzez dodanie drugiej reguły z inną listą warunków. Oprócz reguł wiedzę w systemie wyrażają fakty, które mogą być zadeklarowane wprost przez użytkownika bądź wywnioskowane z zadeklarowanych faktów i reguł.

Reprezentacja wiedzy za pomocą reguł mimo swej pozornej prostoty pozwala na doskonałe opisanie wielu skomplikowanych praktycznych problemów [14], [13]. Zastosowanie takiego sposobu reprezentacji wiedzy pozwala na uzyskanie dużej modularności bazy wiedzy [14], ułatwia przyrostową rozbudowę bazy wiedzy i, co jest bardzo istotne w systemach ekspertowych, pozwala przedstawić wiedzę eksperta w sposób intuicyjny, jasny i przejrzysty oraz łatwy do zweryfikowania. Jest to bardzo istotne podczas budowy i późniejszej eksploatacji i uaktualniania bazy wiedzy, gdzie eksperci dziedzinowi zazwyczaj nie są fachowcami od systemów ekspertowych. W takim przypadku jest im tą wiedzę łatwiej wyrazić a następnie zweryfikować.
Twórcy systemów ekspertowych zazwyczaj bazują na regułach typu:

Wniosek **jeżeli** warunek 1 **i** warunek 2 **i** warunek 3 **i** ...

Regułę można podzielić na dwie części: wniosek i część warunkową. Warunków może być dowolna liczba, wniosek zazwyczaj jest jeden. Reguły takie interpretuje się w ten sposób, że wniosek będzie uznany za prawdę wtedy,

gdy prawdziwe będą wszystkie warunki (oczywiście wtedy gdy warunki połączone są spójnikiem „and"). Gdy w systemie jest kilka reguł o tym samym wniosku to spełnienie warunków jednej z nich wystarcza do uznania wniosku za prawdę. Można zilustrować to przykładem:
reguła1: wniosek **jeżeli** A **i** B
reguła2: wniosek **jeżeli** C **i** D
Gdy prawdziwe jest A i B to wniosek jest prawdziwy, nawet jeżeli C i D nie są prawdziwe. Jeżeli system nie pozwala na stosowanie spójnika „lub" w części warunkowej reguły można go w prosty sposób zastąpić dwiema regułami o tym samym wniosku. Zamiast:
reguła1: A **jeżeli** B **lub** C
można wygenerować dwie reguły:
reguła1: A **jeżeli** B
reguła2: A **jeżeli** C.
Bardzo istotną cechą regułowej reprezentacji wiedzy jest możliwość zagnieżdżania reguł, czyli wykorzystania wniosku jednej reguły jako przesłanki do następnej. Możliwość zagnieżdżania reguł pozwala na czytelniejsze i bardziej przejrzyste wyrażenie wiedzy dziedzinowej, lepiej oddające wiedzę eksperta [16], [26]. Przykład zagnieżdżenia:
reguła1: A **jeżeli** B **i** C
reguła2: C **jeżeli** D **i** E
reguła3: B **jeżeli** G **i** H
Teoretycznie tą samą treść można wyrazić za pomocą jednej reguły:
reguła1: A **jeżeli** D **i** E **i** G **i** H
Jednakże takie wyrażenie nie będzie z jednej strony oddawało pełni treści (traci się możliwość zapytania o prawdziwość B i C) i będzie znacznie mniej czytelne: dla dużej bazy wiedzy reguła mogłaby liczyć kilkadziesiąt bądź kilkaset warunków, dodatkowo w niektórych sytuacjach następuje eksplozja kombinatoryczna i takich „spłaszczonych" reguł mogłaby być ogromna ilość, co wyklucza praktycznie możliwość jakiejkolwiek ich analizy przez dziedzinowych ekspertów. W związku z czym należy wykorzystywać możliwość zagnieżdżenia reguł tak często jak tylko jest to możliwe. W klasycznych systemach ekspertowych fakty, warunki i wnioski reguł są reprezentowane przez zmienne logiczne, które mogą przyjmować wartość prawda lub fałsz. Innym sposobem reprezentacji faktów, warunków i wniosków jest schemat:
Atrybut(Obiekt)=Wartość
Schemat ten bazuje na tym, że człowiek często postrzega świat jako układ obiektów mających jakieś cechy (atrybuty), które przyjmują jakieś wartości np.:

$$\text{Stan finansów(firma X)} = \text{dobry}$$

Wyrażenie to zadeklarowane jako fakt oznacza, że stan finansów firmy X jest dobry. Gdyby zapisać taką regułę:

ocena kredytobiorcy(firma X)=dobra **jeżeli** stan finansów(firma X)=dobry

To można ją tak zinterpretować, że jeśli stan finansów firmy X będzie dobry, to jej ocena jako kredytobiorcy będzie również dobra. Zazwyczaj w tego rodzaju systemach wymagany jest tylko atrybut, w związku z czym można napisać tak:

ocena kredytobiorcy=dobra **jeżeli** stan finansów=dobry

Czyli bez wyszczególniania kogo owa reguła dotyczy. Niektóre systemy pozwalają również na wykorzystywanie zmiennych w regułach w analogiczny sposób jak w języku PROLOG. Reprezentacja faktów, warunków i wniosków za pomocą schematu Obiekt-Atrybut-Wartość dla symbolicznych wartości nie niesie żadnych specjalnych udogodnień w reprezentacji wiedzy, gdyż zamiast np. zapisywać:

Stan finansów(firma X) = dobry

można to samo zadeklarować jako zmienną logiczną:

Stan finansów firmy X jest dobry

Przypisanie jej wartości „prawda" niesie taką samą treść jak w przypadku wykorzystania schematu Obiekt-Atrybut-Wartość. Inaczej jest jeśli system pozwala jako wartość w schemacie wykorzystać wartości numeryczne, wtedy w warunkach reguł można wykorzystać (zazwyczaj systemy na to pozwalają) znaki większości, mniejszości itp. co powoduje, że system staje się nieco bardziej elastyczny i pozwala na reprezentację pewnych (prostych) zależności ilościowych. To jak system wnioskuje dla różnych wartości zostanie wyjaśnione w dalszej części rozdziału podczas opisu mechanizmów wnioskowania.

Czytelnik, który uważnie przeczytał rozdział poświęcony programowaniu w logice może się zastanowić, jaką przewagę daje system ekspertowy w reprezentacji wiedzy nad np. językiem PROLOG: czy aby PROLOG nie jest skuteczniejszy, bardziej elastyczny i dający większe możliwości modelowania zależności świata rzeczywistego niż system ekspertowy z regułową reprezentacją wiedzy? Odpowiedź na to pytanie brzmi: i nie i tak. Nie, ponieważ w systemie ekspertowym można zastosować mechanizm wnioskowania inny od tego jaki jest w PROLOG-u (o tym będzie w części poświęconej mechanizmom wnioskowania), a tak, ponieważ rzeczywiście klasyczne reguły z systemu ekspertowego można (po zazwyczaj drobnej przeróbce składniowej) wykorzystywać w programach języka PROLOG. Nie należy jednak zapomnieć,

że PROLOG to język programowania i za jego pomocą niejednokrotnie budowano właśnie systemy ekspertowe, natomiast system ekspertowy to tylko narzędzie wspomagające podejmowanie decyzji, ekspertyzy itp. Generalnie warto też wiedzieć, że systemy ekspertowe są nieco starsze niż język PROLOG.

Reguły produkcyjne

Zainteresowany czytelnik mógł napotkać w różnych pozycjach poświęconych sztucznej inteligencji termin „reguły produkcyjne" (dosłowne tłumaczenie z angielskiego: „production rules") i niewątpliwie może zadać pytanie o to co owe reguły produkują i jak się mają do reguł znanych z niniejszej książki. Termin ten odnosi się do reguł bardzo zbliżonych do tych klasycznych znanych już czytelnikowi. Różnica tkwi w nieco bardziej „praktycznym", z punktu widzenia informatyka, podejściu do ich działania. Klasyczne reguły mają formę:

jeżeli warunek 1 **i** warunek 2 **i** warunek 3 **i** ... **wtedy** Wniosek

Prawdziwość warunków implikuje w nich prawdziwość wniosku. Reguły produkcyjne można traktować jako pewne rozszerzenie reguł klasycznych. Reguła produkcyjna w rzeczywistości różni się tym, że wnioskiem jej jest (może być) jakaś akcja (lub grupa akcji) a nie, jak to jest w klasycznych regułach, przypisanie wartości logicznej wnioskowi:

jeżeli warunek 1 **i** warunek 2 **i** warunek 3 **i** ... **wtedy** Akcja

Generalnie reguły produkcyjne funkcjonują w systemie, w którym jest mechanizm wnioskowania i tzw. pamięć robocza (ang. working memory). Regułę produkcyjną (podobnie jak i „normalną") można podzielić na dwie części: lewostronną (warunkową) i prawostronną (wniosek, bądź wnioski) W pamięci roboczej przechowywane są obiekty. Mechanizm wnioskowania próbuje sprawdzić, czy obiekty w pamięci roboczej spełniają warunki lewostronnej części reguły, jeśli tak jest to reguła zostaje wykonana (mówi się o „odpaleniu" reguły, termin pochodzi z j. ang.), tzn wykonane są polecenia prawostronnej części reguły. Wśród tych poleceń może być np.: *assert*, które wpisuje nowy obiekt do pamięci roboczej, co można uznać za działanie zbliżone do dodania nowego faktu do pamięci podręcznej znanej z klasycznego systemu ekspertowego, *modify* który pozwala na zmianę obiektu w pamięci roboczej i *remove*, który usuwa obiekt z pamięci roboczej, bądź akcje innego rodzaju. Systemy oparte na regułach produkcyjnych są dość popularne ze względu na ich łatwość w łączeniu ze środowiskami programistycznymi i możliwość pracy z obiektowymi językami programowania. Najpopularniejsze dziś systemy to JESS i DROOLS, szczególnie ten ostatni znakomicie wpisuje się w obiektowe środowisko języka JAVA pozwalając na stosunkowo łatwą integrację z większymi aplikacjami. Oba te systemy w mechanizmie

wnioskowania wykorzystują algorytm RETE, którego zadaniem jest przyspieszenie procesu uzgadniania zawartości pamięci roboczej z warunkami reguł (więcej szczegółów można znaleźć m.in. tu [4].

3.1.2. Reguła zamkniętego świata

Prawdopodobnie każda, nawet bardzo rozbudowana, baza wiedzy nie zawiera wiedzy kompletnej: zawsze może się pojawić sytuacja, zbieg okoliczności czy przypadek, który wykracza poza tematykę sformalizowaną w bazie. Poza tym, w większości sytuacji decyzyjnych, obok wiedzy specjalistycznej (jaką może zawierać baza wiedzy) konieczna jest wiedza zdroworozsądkowa, której prawdopodobnie nie da się w kompletny sposób sformalizować. Taka sytuacja praktycznie uniemożliwia realizację w pełni kompletnego, działającego w jakiejś konkretnej dziedzinie, systemu ekspertowego. Nikt nie jest w stanie zagwarantować, że jakiś szczególny i nie przewidziany wcześniej zbieg okoliczności nie będzie wykraczał poza wiedzę zdefiniowaną w bazie i spowoduje, że system wyda niepoprawny z punktu widzenia merytoryki wniosek. W związku z tym twórcy systemów ekspertowych muszą z góry założyć pewne ograniczenia możliwości realizowanych przez nich narzędzi. Owo ograniczenie nazywa się zasada zamkniętego świata i mówi, że:
Wszystko co nie zostało zadeklarowane wprost, bądź nie zostało wywnioskowane z reguł traktuje się jako nieprawdę
Czyli jeśli do bazy wiedzy dotyczącej analizy zdolności kredytowej przedsiębiorstw zadeklarowano fakty dotyczące stanu finansów przedsiębiorstwa, jego zadłużenia, pozycji na rynku, itp. a nie zadeklarowano nic na temat przeszłości kryminalnej jego właścicieli przyjmuje się, że takowej oni nie mieli.

3.1.3. Warunki wzajemnie wykluczające się

Podczas budowy bazy wiedzy niejednokrotnie zdarza się, że pewne warunki mogą się wzajemnie wykluczać, np.: „znajomość języka angielskiego" i „brak znajomości języka angielskiego". Nie mogą one (chyba, że dotyczą innych osób) być jednocześnie prawdziwe, tzn. gdy jeden z nich jest prawdziwy to drugi automatycznie jest nieprawdziwy. Aby system podczas wnioskowania nie zadawał absurdalnych (z ludzkiego punktu wiedzenia) pytań oraz by wykluczyć deklarację niezgodnych ze zdrowym rozsądkiem sytuacji, sposób reprezentacji wiedzy w systemie powinien uwzględniać takie sytuacje, tzn. powinien pozwalać na deklarację, grup wzajemnie wykluczających się warunków. Mechanizm wnioskowania w takiej sytuacji powinien rozpoznawać tego rodzaju sytuacje i gdy użytkownik zadeklaruje, że jeden z warunków

jest prawdziwy (np. „brak znajomości języka angielskiego") to reszta warunków z tej listy wzajemnie wykluczających się powinna automatycznie zostać uznana za fałsz (np. „znajomość języka angielskiego").

3.2. Mechanizm wnioskowania

System ekspertowy składa się przede wszystkim z dwóch podstawowych części: bazy wiedzy i mechanizmu wnioskowania. W poprzednim rozdziale omówione zostały najważniejsze sposoby reprezentacji wiedzy, teraz autor proponuje przyjrzeć się temu jak działają mechanizmy wnioskowania w systemach ekspertowych. Generalnie wyróżnia się dwa podstawowe rodzaje mechanizmów wnioskowania:

— Wnioskowanie w przód (zwane czasem wnioskowaniem progresywnym),
— Wnioskowanie wstecz (zwane czasem wnioskowaniem regresywnym).

Niektórzy autorzy wymieniają również wnioskowanie mieszane, będące połączeniem dwóch poprzednich mechanizmów ale nie ma ono większego zastosowania w praktyce.

3.2.1. Wnioskowanie w przód

Poniżej zostanie omówiony proces wnioskowania w przód, przedstawiony zostanie dla najprostszego rodzaju regułowej bazy wiedzy, gdzie warunkami i wnioskami reguł są zmienne logiczne, bez możliwości zaprzeczeń warunków. Pierwszym krokiem do opisu mechanizmu wnioskowania w przód jest zdefiniowanie założeń oraz celu wnioskowania. Założenia:

— System ma zdefiniowaną bazę wiedzy, czyli zbiór reguł oraz, ewentualnie, zadeklarowane fakty,
— Fakty (te z bazy wiedzy oraz zadeklarowane bezpośrednio przez użytkownika w trakcie konwersacji) przechowywane są w pamięci podręcznej.

Cel: Celem działania mechanizmu wnioskującego jest znalezienie wszystkich możliwych do osiągnięcia wniosków. Mechanizm wnioskowania w przód działa w następujący sposób:

Proces wnioskowania rozpoczyna się od tego, że system sprawdza, czy warunki pierwszej reguły są spełnione (prawdziwe) sprawdzając, czy są zadeklarowane w pamięci podręcznej. Jeśli wszystkie warunki reguły są spełnione, wniosek reguły uznany zostaje za prawdę i dodany do pamięci podręcznej, jeśli fakty zadeklarowane w pamięci podręcznej nie wypełniają warunków reguły jest ona pomijana i system sprawdza kolejną regułę bazy wiedzy. Jeśli system dojdzie do końca reguł w bazie rozpoczyna sprawdzanie od początku, aż do momentu, w którym nie da się wyprowadzić więcej wniosków. Wnioskowanie w przód można zilustrować poniższym przykładem:

W systemie zadeklarowana bazę wiedzy:

```
reguła 1: A jeżeli B, C
reguła 2: B jeżeli D, E
reguła 3: C jeżeli F
reguła 4: E jeżeli K, L
reguła 5: B jeżeli M, N
reguła 6: N jeżeli L
```

Oraz następujące fakty w pamięci podręcznej:
L, M, F, D. Proces wnioskowania rozpoczyna się od sprawdzenia, czy prawdziwe są warunki reguły 1, ponieważ ani B ani C nie są prawdą, w związku z tym system sprawdza kolejną regułę, której warunki również nie są spełnione. Warunkiem reguły 3 jest F, której jest zadeklarowane w pamięci podręcznej, w związku z czym warunki reguły 3 są spełnione i jej wniosek czyli C można uznać za prawdziwy i zostaje dopisany do pamięci podręcznej, której zawartość wygląda tak:
L, M, F, D, C
System sprawdza regułę 4, której warunkami są L i K. W pamięci jest zadeklarowane L, natomiast nie jest zadeklarowane K, w związku z czym warunki reguły nie są spełnione (warunki są połączone spójnikiem i). Warunkami kolejnej reguły są M i N, W pamięci podręcznej zadeklarowane jest tylko M, w związku z czym system pomija ją i sprawdza ostatnią regułę, której warunkiem jest L. Ponieważ L jest prawdą to wniosek reguły, czyli N również będzie prawdą i zostaje dodany do pamięci podręcznej:
L, M, F, D, C, N
Ponieważ proces doszedł do ostatniej reguły to wraca z powrotem do pierwszej próbując po raz kolejny sprawdzić prawdziwość jej warunków. Reguła będzie spełniona jeśli prawdą będzie B i C, jak na razie jednak prawdą jest tylko C, w związku z czym system przechodzi do kolejnej reguły, której warunków znów nie udało się spełnić, następnie pomija regułę 3 (która już była wykonana) i sprawdza regułę 4, której warunki dalej pozostały nie spełnione. Warunki reguły 5, czyli M i N są spełnione, w związku z czym prawdziwy jest jej wniosek, czyli B, który zostaje dodany do pamięci podręcznej:
L, M, F, D, C, N, B
Reguła 6 jest pomijana i system po raz kolejny wraca do reguły 1, gdzie tym razem warunki okazują się spełnione, wniosek, czyli A zostaje uznany za prawdę i dodany do pamięci podręcznej:
L, M, F, D, C, N, B, A
W dalszej kolejności system próbuje spełnić warunki następnych, dotychczas nie wykorzystanych reguł, których jednak nie da się już uzgodnić. Jeżeli kolejne sprawdzenie całej bazy nie przyniesie żadnych zmian mechanizm kończy pracę podając listę faktów:

L, M, F, D, C, N, B, A

3.2.2. Wnioskowanie wstecz

Celem wnioskowania wstecz nie jest, jak w przypadku wnioskowania w przód, znalezienie wszystkich prawdziwych (możliwych do udowodnienia) faktów a jedynie sprawdzenie, czy jeden, konkretny fakt (hipoteza) jest prawdziwy.

Algorytm wnioskowania wstecz jest swego rodzaju „przeciwieństwem" algorytmu wnioskowania w przód. Celem wnioskowania jest udowodnienie pewnej hipotezy, w związku z czym zanim proces wnioskowania się rozpocznie konieczne jest zadeklarowanie hipotezy. Dla najprostszego przypadku, gdy warunkami i wnioskami reguł są zmienne logiczne postawienie hipotezy polega na sprawdzeniu, czy dany fakt jest prawdziwy. Jak wygląda zatem proces wnioskowania i sprawdzania prawdziwości hipotezy?

W pierwszym kroku algorytm bada, czy postawiona hipoteza nie jest wprost zadeklarowana jako fakt w pamięci podręcznej. Jeśli tak, to zostaje ona potwierdzona i algorytm kończy się sukcesem. Jeśli hipoteza nie jest zadeklarowana jako fakt, to algorytm sprawdza czy w bazie wiedzy są reguły, których wnioskiem jest postawiona hipoteza. Jeśli nie ma takich reguł algorytm kończy pracę klęską, nie mogąc udowodnić hipotezy. W przeciwnym przypadku (gdy są reguły, których wnioskiem jest hipoteza) algorytm próbuje warunki tej reguły potwierdzić, traktując je jako kolejne hipotezy. W sytuacji, gdy potwierdzenie jednej z hipotez zawiedzie, system próbuje znaleźć inne reguły, których wnioskiem jest badana hipoteza, próbując udowodnić jej prawdziwość bazując na innej regule. Dopiero jeśli próba zawiedzie dla wszystkich reguł algorytm kończy się klęską. Tego rodzaju mechanizm wracania i szukania kolejnych reguł nosi nazwę mechanizmu nawrotów. Uważny czytelnik zauważy niewątpliwie, że mechanizm ten, jak i cały algorytm jest bardzo zbliżony do algorytmu standard bakctracking wykorzystywanego w języku PROLOG.
Poniżej jest przedstawiony przykład wnioskowania wstecz. Bazuje on na bazie wiedzy (regułach i faktach) z poprzedniego przykładu a hipotezą jest A, w związku z czym celem wnioskowania jest sprawdzenie czy A jest prawdą: W pierwszym kroku algorytm sprawdza, czy A nie jest aby wprost zadeklarowane w pamięci podręcznej, ponieważ nie jest, to algorytm szuka regułę, której wnioskiem jest A. Wnioskiem reguły 1 jest właśnie A, w związku z czym system stawia sobie B i C (warunki reguły 1) jako kolejne hipotezy:

```
reguła 1: A jeżeli B, C
```

Hipotezy B i C
Ponieważ B nie jest zadeklarowane w pamięci podręcznej system szuka regułę, której wnioskiem jest B. Wnioskiem reguły 2 jest właśnie B a jej warunki t.j. D i E stają się hipotezami:
`reguła 2: B jeżeli D, E`
Hipotezy: D, E, C
D jest zadeklarowane w pamięci podręcznej, w związku z czym jest potwierdzone, natomiast E można wyznaczyć z reguły 4. Warunkami tej reguły są K i L.
`reguła 4: E jeżeli K, L`
Hipotezy: D – potwierdzona, K, L, C
Niestety o ile L jest w pamięci podręcznej o tyle K nie ma ani w pamięci podręcznej ani nie jest wnioskiem żadnej reguły, w związku z czym reguła 4 jest niepotwierdzona.
Hipotezy: D – potwierdzona, K – zawodzi, L – potwierdzona, C
System sprawdza, czy jest jeszcze jakaś reguła, z której można wyznaczyć E. Ponieważ nie ma takiej reguły, poszukiwanie E zawodzi i system sprawdza, czy jest jeszcze jakaś reguła, której wnioskiem jest B. Wnioskiem reguły 5 jest B, w związku z czym jej warunki, czyli M i N stają się hipotezami.
`reguła 5: B jeżeli M, N`
Hipotezy: M, N, C
M jest wprost zadeklarowane w pamięci podręcznej, natomiast N można wyznaczyć z reguły 6. Jej warunek, czyli L jest zadeklarowany w pamięci podręcznej
`reguła 6: N jeżeli L`
Hipotezy: M – potwierdzone, L – potwierdzone, C
W związku z powyższym z reguły 5 system wnioskuje, że B jest prawdą i może zostać dodane do pamięci podręcznej.
Hipotezy: B – potwierdzone, C
W dalszej kolejności system sprawdza, czy drugi warunek reguły 1, czyli C jest prawdą. Ponieważ nie jest zadeklarowany wprost, system szuka reguły, której wnioskiem będzie C. Taką regułą jest reguła 3, jej warunkiem jest L, które jest zadeklarowane w pamięci podręcznej dlatego L zostaje potwierdzone i dodane do pamięci podręcznej.
`reguła 3: C jeżeli F`
Hipotezy: B – potwierdzone, L – potwierdzone
Bazując na regule 1 system potwierdza, że A, czyli główna hipoteza jest prawdą.

3.3. Anomalie

Realizacja dużych regułowych baz wiedzy zazwyczaj nie jest zadaniem prostym. Można w trakcie jego realizacji popełnić szereg błędów, które mogą pogorszyć sprawność działania systemu, spowodować, że będzie on dawał niepoprawne wnioski, bądź w ogóle uniemożliwią wnioskowanie.
Poza błędami ściśle merytorycznymi (które trudno byłoby tu analizować), twórca bazy wiedzy może popełnić generalnie dwa rodzaje błędów:

— Nadmiarowości - nie mające charakteru krytycznego, które nie powodują zawieszenia się procesu wnioskowania,
— Sprzeczności - mające charakter krytyczny, które mogą doprowadzić do zawieszenia procesu wnioskowania, bądź uniemożliwić wykorzystanie niektórych reguł we wnioskowaniu.

Większość praktycznych realizacji systemów ekspertowych ma wbudowane mechanizmy kontroli sprzeczności, niektóre mają również mechanizmy kontroli nadmiarowości. Niestety skuteczność tych mechanizmów (szczególnie kontrolujących nadmiarowości) nie zawsze jest idealna, jednakże pozwalają one wykryć większość anomalii w bazie.

3.3.1. Nadmiarowości

Nadmiarowości nie mają charakteru krytycznego, jednakże zaśmiecają bazę wiedzy niepotrzebnymi regułami, spowalniając proces wnioskowania, przy niekorzystnym układzie reguł mogą doprowadzić do wyprowadzania błędnych wniosków.
Najprostsza nadmiarowość wynika ze zdublowania reguł:

```
reguła 1: A jeżeli B, C
reguła 2: A jeżeli C, B
```

Powyższe reguły różnią się jedynie kolejnością warunków i jedna z nich może zostać bez szkody dla systemu usunięta. Nie zawsze znalezienie zdublowanych reguł jest takie proste, gdyż mogą one być ukryte w różnych miejscach bazy i na różnych poziomach zagnieżdżeń, np.:

```
reguła 1: A jeżeli B, C
reguła 2: B jeżeli D, E
reguła 3: C jeżeli F, G
reguła 4: A jeżeli D, E, F, G
```

Powyższy przykład pokazuje, że reguła 4 dubluje regułę 1 powtarzając ją w wersji spłaszczonej (bez zagnieżdżeń). W takim wypadku oczywiści powinno się jedną z tych reguł usunąć. W tego rodzaju sytuacjach zazwyczaj lepiej jest usunąć regułę, którą jest spłaszczoną wersją kilku innych, gdyż zagnieżdżenia po pierwsze czynią bazę czytelniejszą, łatwiejszą do kontroli i mniejszą a po drugie lepiej odwzorowują strukturę wiedzy niż reguły płaskie.

Inny rodzaj nadmiarowości wynika z tzw. subsumpcji reguł, np.:
`reguła 1: A jeżeli B`
`reguła 2: A jeżeli B, C`
W przypadku takich reguł, jeśli prawdziwe będzie B, to A będzie prawdziwe niezależnie od tego czy prawdziwe będzie C, w związku z czym reguła 2 jest niepotrzebna. Tego rodzaju anomalie najczęściej powstają podczas nieuważnej aktualizacji bazy, gdy ekspert dodaje kolejną, ostrzejszą regułę (w przykładzie reguła 2) a nie usunie „starej", bardziej łagodnej reguły (w przykładzie reguła 1). W takiej sytuacji pomimo dodania nowej reguły system i tak będzie wnioskował wg starej. Podobnie jak w przypadku zdublowanych reguł subsumpcja może być ukryta między kilkoma zagnieżdżającymi się regułami.

Kolejny rodzaj nadmiarowości może wynikać z wykorzystania warunków wzajemnie wykluczających się, np. Gdy warunki B i C wzajemnie się wykluczają (jeśli B jest prawdziwe to C jest nieprawdziwe i na odwrót, zawsze jeden jest prawdziwy a drugi nie) i baza wiedzy wygląda tak:
`reguła 1: A jeżeli B, D`
`reguła 2: A jeżeli C, D`
Niezależnie od tego, który z tych wykluczających się atrybutów będzie prawdą o prawdziwości wniosku decydować będzie prawdziwość warunku D.

3.3.2. Sprzeczności

Sprzeczności w przeciwieństwie do nadmiarowości mają charakter krytyczny, są znacznie „groźniejsze" dla bazy wiedzy i procesu wnioskowania. Przede wszystkim sprzeczność w bazie wiedzy prowadzić może bądź do całkowitego „zawieszenia" procesu wnioskowania, bądź do tego, że wybrane reguły nigdy nie będą mogły być spełnione.
Na marginesie warto wspomnieć, że pod terminem sprzeczności, niekoniecznie ukrywają się tu sprzeczności w sensie logicznym. Termin ten w przypadku systemów ekspertowych odnosi się do specyficznych anomalii w bazie wiedzy.

Jakiego rodzaju sprzeczności mogą się w bazie pojawić?
Najprostszym rodzajem sprzeczności jest sytuacja, w której warunek reguły jest jednocześnie jej wnioskiem, np.:
`reguła 1: A jeżeli A, B`
Taka reguła nigdy nie będzie prawdziwa a gdy mechanizm wnioskowania wstecz będzie chciał uzgodnić A, to wpadnie w nieskończoną pętlę, tzn. gdy Hipotezą będzie A, to jej jednym z jej warunków będzie A i chcą go uzgodnić rekurencyjnie odwoła się znów do tej samej reguły itd.

Podobnie jak nadmiarowość, sprzeczność może być ukryta na między regułami na różnych poziomach zagnieżdżenia, np.:

```
reguła 1: A jeżeli B, C
reguła 2: B jeżeli D, E
reguła 3: C jeżeli F
reguła 4: D jeżeli A
```

Nawet jeśli na pierwszy rzut oka sprzeczność nie jest widoczna, to gdyby dokonać spłaszczenia reguły 1 (tzn. zlikwidować zagnieżdżenia reguły) to wtedy miałaby ona taką postać:

```
reguła 1: A jeżeli A, E, F
```

W takim kształcie jest już ona widoczna wyraźnie. Najprostszym sposobem znajdywania tego rodzaju sprzeczności jest właśnie dokonywanie spłaszczeń poszczególnych reguł i sprawdzanie, czy w trakcie któregoś z nich nie pojawi się sytuacja, w której warunek reguły stanie się jej wnioskiem.

Inny rodzaj sprzeczności wynika z wykorzystania wzajemnie wykluczających się warunków, tzn. gdy oba wzajemnie wykluczające się warunki będą w jednej regule to nigdy nie będzie mogła być spełniona, np.:
B, C – warunki wzajemnie wykluczające się,

```
reguła 1: A jeżeli B, C
```

W takiej sytuacji, gdy B jest prawdziwe, to C zawsze jest nieprawdą, gdy C jest prawdziwe, to B na pewno będzie nieprawdą. Taki przypadek sprzeczności może, podobnie jak powyższy, ukryć się między kilkoma zagnieżdżonymi regułami, np.:
D, F – warunki wzajemnie wykluczające się,

```
reguła 1: A jeżeli B, C
reguła 2: B jeżeli D, E
reguła 3: C jeżeli F, G
```

Znów na pierwszy rzut oka nie widać sprzeczności ale spłaszczenie pokazuje, że taka reguła nigdy nie będzie spełniona:

```
reguła 1: A jeżeli D, E, F, G
```

Powyższe przykłady nadmiarowości i sprzeczności nie wyczerpują szerokiego zakresu konfliktów między regułami jakie mogą zajść z powodów merytorycznych.

3.4. Wnioskowanie niemonotoniczne

Uważny czytelnik może zauważyć, że zaprezentowany powyżej sposób reprezentacji wiedzy regułowej i wnioskowania ma poważny mankament: nie pozwala na stosowanie zaprzeczeń w części warunkowej, tzn. o ile można zdefiniować regułę, która mówi, że:

```
reguła 1: dać kredyt jeżeli dobre finanse i dobra reputacja
```
o tyle regułę w takiej postaci:
```
reguła 1: dać kredyt jeżeli dobre finanse, dobra reputacja, nie
karany właściciel
```
nie można wyrazić bezpośrednio. Można oczywiście zdefiniować dwa wykluczające się warunki: „karany właściciel" i „nie karany właściciel" jest to jednak po pierwsze „nieeleganckie" rozwiązanie, a po drugie ogranicza ono mocno elastyczność systemu wymuszając konieczność definiowania reguł dla wszystkich przypadków i ich zaprzeczeń.
Odpowiedzią na tego rodzaju ograniczenie jest oczywiście dodanie do mechanizmu wnioskowania umiejętności rozpoznawania i odpowiedniego posługiwania się funktorem „nie".

3.4.1. Zaprzeczenia

Na wstępie warto rozważyć jak interpretować owo zaprzeczenie. Aby połączenie klasycznych reguł z negacją przeszło bezboleśnie, konieczne jest precyzyjne zdefiniowanie owego zaprzeczenia, szczególnie w kontekście reguły zamkniętego świata.

W klasycznym, omówionym wcześniej, rozwiązaniu jeśli jakiś fakt nie został zadeklarowany wprost i nie dał się wywnioskować z reguł uznawany był za fałsz. W systemie pozwalającym na zaprzeczanie warunków należy wprowadzić możliwość zadeklarowania, że jakiś fakt na pewno nie jest prawdą, czyli np. nie A.
Jeśli w systemie jest reguła klasyczna, czyli bez zaprzeczonych warunków, to system wnioskuje tak samo jak klasyczny, gdy zaś analizowana reguła zawiera zaprzeczony warunek, np.:
```
reguła 1: A jeżeli B, nie C
```
wtedy A będzie prawdziwe wtedy, gdy B będzie prawdą i C będzie wprost zadeklarowane jako fałsz albo system nie będzie mógł udowodnić, że C jest prawdą.
Bazując na tym założeniu można spróbować przyjrzeć się jak będzie działał mechanizm wnioskowania wstecz dla przykładowej bazy wiedzy:
```
reguła 1: A jeżeli B, C
reguła 2: B jeżeli D, E
reguła 3: C jeżeli nieF
reguła 4: E jeżeli K, L
reguła 5: B jeżeli nieM, N
reguła 6: N jeżeli L
reguła 7: C jeżeli nieK
```

Oraz następujące fakty w pamięci podręcznej:
L, nieM, F, D, nieK.
Hipotezą wnioskowania jest A.
Ponieważ ani A ani B i C nie są zadeklarowane w pamięci podręcznej system sprawdza, czy B i C są wnioskami reguł. B jest wnioskiem reguły 2, w związku z czym system sprawdza, czy jej warunki, czyli D i E są prawdziwe. ponieważ D jest zadeklarowane w pamięci podręcznej to jest uznane za prawdę, natomiast E jest wnioskiem reguły 4. Warunkami reguły 4 są L i K. L jest prawdą, ale w pamięci podręcznej zadeklarowano wprost, że K jest nieprawdą. W związku z tym, nie mogąc potwierdzić B z reguły 2 system poszukuje kolejne reguły, której wnioskiem jest B. Warunkami reguły 5 są nieM i N. Ponieważ nieM zostało zadeklarowane wprost a N można wywnioskować z reguły 6, to ostatecznie B można uznać za prawdę.
Reguła 3 mówi, że C będzie prawdą, gdy nieprawdą będzie F, ponieważ F jest prawdą, to system nie może potwierdzić C, w związku z czym w następnym kroku próbuje udowodnić C korzystając z reguły 7. Warunkiem reguły 7 jest nieK, które zostało wcześniej wywnioskowane i dodane do pamięci podręcznej, w związku z czym C zostało potwierdzone oraz w dalszej kolejności z reguły 1 zostało potwierdzone A.

Realizacja wnioskowania w przód dla powyższej bazy wiedzy jest chyba jeszcze ciekawsza. Dysponując następującymi, zadeklarowanymi wprost faktami w pamięci podręcznej:
L, nieM, F, D.
Mechanizm wnioskowania rozpoczyna od próby podstawienia faktów do reguł. Ponieważ brak jest danych dla Reguły 1 oraz reguły 2, system próbuje podstawić istniejące fakty do reguły 3:
`reguła 3: C jeżeli nieF`
Ponieważ w pamięci zadeklarowano F, to wnioskiem z reguły jest nieC. Wniosek ten został dodany do pamięci podręcznej. W kolejnym kroku system próbuje uzgodnić warunki kolejnych reguł i udaje się to dla reguły 6:
`reguła 6: N jeżeli L`
ponieważ prawdą jest L, to prawdą jest również N, które zostaje dodane do pamięci podręcznej. Kolejną regułą, na bazie której system może wnioskować jest reguła 7:
`reguła 7: C jeżeli nieK`
Ponieważ zostało zadeklarowane, że K jest nieprawdą, to w związku z tym C jest prawdą i zostaje dodane do pamięci podręcznej:
L, nieM, F, D, nieC, N, C
Nie jest trudno zauważyć niespójność: w pamięci podręcznej zadeklarowane jest nieC i C. Ponieważ taki przypadek jest naruszeniem zdrowego rozsądku (oraz zasady wyłączonego środka), to system musi sobie jakoś z tym

poradzić. Radzi sobie bazując na zasadzie, wg której jeśli z chociaż jednej reguły wynika, że dany wniosek jest prawdą, to mimo, że z innych może wynikać, że jest nieprawdą traktuje się go jak prawdę. Gdyby dołożyć kolejną regułę mówiącą, że badany wniosek jest nieprawdą to nie zmieni to podjętego wcześniej wniosku. Zmieniony by został tylko wtedy, gdyby reguła z której wynikałoby, że jest prawdą przestała obowiązywać. W związku z tym zawartość pamięci podręcznej wyglądać będzie tak:
L, nieM, F, D, N, C
Próby dalszego podstawiania zadziałają dla reguły 5:
`reguła 5: B jeżeli nieM, N`
której wnioskiem będzie B. Mając dane B i C z reguły 1 można wywnioskować, że prawdą jest A. Więcej z takiej bazy wiedzy wywnioskować nie można.
Analizując powyższy proces wnioskowania można zauważyć jedną charakterystyczną właściwość, której nie było przy klasycznym sposobie wnioskowania: coś raz uznane za nieprawdę mogło w trakcie wnioskowania zmienić się w prawdę i odwrotnie (przykład tego nie pokazywał ale można takowy znaleźć np. tu [16]). Taka właściwość nazywa się niemonotonicznością i oznacza, że raz wydedukowana wartość logiczna wniosku w trakcie wnioskowania może zostać zmieniona. Jest to ważny aspekt procesu rozumowania i występuje jak najbardziej w zdroworozsądkowym rozumowaniu jakim posługuje się człowiek (nie bez przyczyny mówi się, że tylko krowa nie zmienia poglądów).
Warto zwrócić uwagę na to, że możliwość dodawania zaprzeczeń do reguł wymusza podobną jak w klasycznych systemach ekspertowych konieczność czuwania nad unikaniem sprzeczności między regułami, które to mogą powstawać także dzięki wykorzystaniu zaprzeczeń np.:
`reguła 1: A jeżeli B, nieB`
Mechanizm powstawania sprzeczności jest tu analogiczny do mechanizmu w klasycznym, monotonicznym systemie.

Generalnie o monotoniczności mówi się wtedy, gdy dodanie nowej wiedzy do bazy wiedzy nie spowoduje falsyfikacji faktów już wywnioskowanych [4]. Odnosząc to do opisywanych wcześniej sposobów reprezentacji wiedzy bez trudu można zauważyć, że klasyczna regułowa reprezentacja wiedzy, bez możliwości zaprzeczania warunków jest monotoniczna (dodanie nowego faktu bądź reguły nie może spowodować, że żadna wcześniej wywnioskowana konkluzja nie zostanie podważona). Dodanie możliwości zaprzeczania warunków spowoduje, że w pewnych sytuacjach coś raz uznane za prawdę, po dodaniu nowego faktu, bądź reguły zostanie uznane za fałsz.

Charakterystyczną cechą wnioskowania niemonotonicznego jest to, że

dodanie nowej wiedzy do bazy wiedzy może prowadzić do niespójności całej bazy, co więcej połączenie dwóch spójnych baz wiedzy może również doprowadzić do niespójności.

Kolejna ciekawa kwestia związana z wnioskowaniem niemonotonicznym i spójnością wnioskowania to sprawa zagnieżdżeń reguł z możliwością zaprzeczeń. Gdyby pozwolić na zaprzeczenie także wniosków reguł może to doprowadzić do powstawania konfliktów w bazie, które mogą wyniknąć z różnych ścieżek dojścia do zaprzeczających sobie wniosków i wprost do złamania zasady wyłączonego środka. Zilustrować można to przykładem takiej bazy wiedzy:

```
reguła 1: Ssak jeżeli Delfin
reguła 2: nieStworzenie morskie jeżeli Ssak
reguła 3: Stworzenie morskie jeżeli Delfin
```

Mając taką bazę wiedzy i deklarując, że coś jest delfinem, to wnioskiem końcowym jest jednocześnie to, że nie jest on stworzeniem morskim (z reguły 1 i 2) oraz jest stworzeniem morskim (z reguły 3). Jak w takim razie sobie powinien radzić mechanizm wnioskowania? Przy tego rodzaju problemach najprostszym rozwiązaniem jest wykorzystanie tzw. najkrótszej drogi, tzn. wybranie takiej ścieżki wnioskowania, dla której jest najmniejsza liczba reguł (węzłów). w takim wypadku, dla analizowanego przykładu rozwiązaniem jest wykorzystanie tylko reguły 3, co odpowiada wnioskowaniu zdroworozsądkowemu. Proces wnioskowania może w takim razie wyglądać tak:

1. sprawdzenie, czy nie powstają konflikty
2. wybranie najkrótszej ścieżki dla konfliktowych wniosków
3. wnioskowanie

Niestety mechanizm ten nie zawsze działa dobrze: po pierwsze w przypadku bardzo mocno zagnieżdżonych baz wiedzy różnica w ilości reguł może być nieistotna (np. wg jednej ścieżki 155 reguł a wg drugiej 154 reguły), bądź w sytuacji, gdy w bazie wiedzy są nieuzasadnione „skróty" np.:

```
reguła 1: Ssak jeżeli Waleń
reguła 2: nieStworzenie morskie jeżeli Ssak
reguła 3: Stworzenie morskie jeżeli Waleń
reguła 4: Waleń jeżeli Delfin
reguła 5: Delfin jeżeli Delfin długopyski
reguła 6: Ssak jeżeli Delfin długopyski
```

Powyższy przykład pokazuje, że najkrótsza ścieżka od delfina długopyskiego prowadzi przez regułę 6 i 2 ale wniosek z niej jest błędny. Poprawny wniosek system wyciągnie dla reguł 5, 4, 3. Błąd wynika z niepotrzebnego „skrótu" wnioskowania w regule 6. To, że delfin długopyski jest ssakiem można wywnioskować z reguł 5, 4, 1.

3.4.2. Wnioskowanie na bazie założeń

Innym charakterystycznym rodzajem wnioskowania niemonotonicznego jest wnioskowanie na bazie założeń (default reasoning). Idea wnioskowania na bazie założeń opiera się na tym, że w potocznym rozumowaniu często człowiek korzysta z zestawu domyślnych założeń, reguł i stwierdzeń, które jeśli nie zostały wprost podważone uważane są za prawdę. Przykładem może być stwierdzenie: „ptaki latają". Stwierdzenie takie jest generalnie prawdą, ale gdy mowa jest o konkretnym ptaku o imieniu Tweety, który jest pingwinem, to ten, dla analizowanej konkretnej sytuacji, podważa powyższe stwierdzenie. Podważa je dlatego, że pingwiny są takim rodzajem ptaków, które nie latają, czyli są wyjątkiem od ogólnej zasady. Powyższy przykład można sformalizować korzystając z języka logiki:

$$\forall x(\text{ptak}(X) \rightarrow \text{lata}(X)).$$
$$\forall x(\text{pingwin}(X) \rightarrow \text{ptak}(X)).$$
$$\forall x(\text{pingwin}(X) \rightarrow \neg\text{lata}(X)).$$

Taki zapis jest oczywiście sprzeczny, ponieważ nie funkcjonuje w nim zasada wyłączonego środka: pingwin lata bo jest ptakiem i jednocześnie nie lata, bo jest pingwinem. chcąc powyższy model zrobić niesprzecznym należałoby go zapisać tak:

$$\forall x(\text{ptak}(X) \wedge \neg\text{pingwin}(X) \rightarrow \text{lata}(X)).$$
$$\forall x(\text{pingwin}(X) \rightarrow \text{ptak}(X)).$$
$$\forall x(\text{pingwin}(X) \rightarrow \neg\text{lata}(X)).$$

Powyższy model nie wyczerpuje rzecz jasna wszystkich wyjątków od reguły, bo nie tylko pingwiny są nielatającymi ptakami.
Podobny przykład często pojawia się w literaturze ([4] i inne) zwany jest paradoksem Nixona i wynikał on z tego, że Nixon deklarował się jako kwakier i był prezydentem z ramienia Partii Republikańskiej. Jako kwakier powinien być pacyfistą a jako republikanin pacyfistą być nie powinien. Paradoks obu powyższych przykładów wynikał z tego, że istnieje ogólna zasada („ptaki latają", „kwakrzy to pacyfiści"), którą przy braku dowodów podważających uznaje się za prawdę. Pojawienie się konkretnego przypadku podważającego ogólną zasadę powoduje, że zostaje ona dla niego zniesiona. Takie wnioskowanie jest zaburzeniem zasady zamkniętego świata, gdyż przy braku dowodów potwierdzających (ale i braku zaprzeczających), domyślnie uznaje się coś za prawdę (np. Tweety lata).
Problematykę wnioskowania na bazie założeń, formalizuje tzw. logika domyślna [4] (bardzo nieszczęśliwe tłumaczenie angielskiego terminu default

logic). Bazuje ona na klasycznej logice pierwszego rzędu ale wprowadza dodatkowe pojęcie reguły domyślnej, w postaci:

$$\alpha : \beta / \delta$$

gdzie α to warunek, β to warunek spójności, natomiast δ to wniosek. Interpretuje się ją tak, że jeśli prawdziwy jest warunek i nie jest naruszony w bazie wiedzy warunek spójności, to wniosek reguły jest prawdziwy.
Powyższy przykład z ptakiem można opisać tak:

$$\text{ptak(X)} : \text{lata(X)} / \text{lata(X)}$$

Czyli jeżeli ktoś jest ptakiem i nie jest zadeklarowane, że nie lata to lata.
Dla paradoksu Nixona sytuacja jest bardziej skomplikowana, gdyż pojawia się tam sprzeczność między dwoma domyślnymi regułami: „kwakier to pacyfista" i „republikanin to nie pacyfista". Problem konfliktów między domyślnymi regułami jest dość szeroko omówiony w literaturze i ciągle jest otwarty, próbuje się go rozwiązać za pomocą nadawania priorytetów regułom, próbie oceny ich szczegółowości i wykorzystania zasady znoszenia reguły ogólniejszej przez regułę bardziej szczegółową, bądź wyboru tylko tych reguł, dla których wnioskowanie jest spójne. Dla przykładu z Nixonem jednak najlepsze, chyba, jest zwrócenie uwagi na to jak ten konflikt zostałby rozwiązany przez człowieka. W potocznym rozumowaniu, stojąc przed tego rodzaju konfliktem należy zwrócić uwagę na to, że członek Partii Republikańskiej to polityk, a polityk to nie taki „normalny" kwakier i, być może, zasady moralne jakie obowiązują „zwykłych" kwakrów nie traktuje on zbyt rygorystycznie. Trzymając się tychże założeń można przedstawić taki model:

$$\text{republikanin(X)} \rightarrow \neg\text{pacyfista(X)}$$

$$\text{republikanin(X)} \rightarrow \text{polityk(X)}$$

$$\text{kwakier(X)} : (\text{pacyfista(X)} \wedge \neg\text{polityk(X)}) / \text{pacyfista(X)}$$

Powyższy model zawiera dwie „zwykłe" implikacje i jedną domyślną regułę. Dodanie faktów mówiących, że ktoś jest jednocześnie republikaninem i kwakrem nie będzie powodować wtedy konfliktów.

3.4.3. Inne rodzaje wnioskowania niemonotonicznego

Wnioskowanie na bazie założeń jest tylko jednym z przykładów wnioskowania niemonotonicznego. Logika autoepistemiczna np. zakłada istnienie operatora przekonania $\Box$ o tym, że analizowana formuła jest prawdą. Wyrażenie: $\Box X$ oznacza, że wiadomo, iż X jest prawdą, wyrażenie: $\Box\neg X$ oznacza, że wiadomo, iż X nie jest prawdą, zaś wyrażenie $\neg\Box X$ oznacza, że

nie wiadomo nic o X.
W praktycznych realizacjach systemów wykorzystujących wnioskowanie niemonotoniczne popularne są systemy podtrzymujące prawdę (Truth Maintenance System) m.in. JTMS bazujący na wykorzystaniu operatora „zgodny z ...” i dwoma zbiorami przekonań: IN (opisujący zdania, które muszą być spełnione aby wniosek była prawdziwy) i OUT (opisujący zdania, które muszą nie być spełnione), oraz system ATMS, który bazuje na zdaniach pewnych i przypuszczeniach, które mogą ale nie muszą być prawdą.

Problematyka logiki i wnioskowania niemonotonicznego jest ciągle mocno rozwijana, szczególnie w kontekście wykorzystania ich w systemach wnioskowania dla systemów działających w zakresie prawa, gdyż tam bardzo często z tego rodzaju rozumowaniem można się spotkać.

3.5. Wiedza niepełna i niepewna

Proces wnioskowania w rzeczywistych sytuacjach bardzo rzadko opiera się na pełnej wiedzy na temat warunków i reguł opisujących analizowany problem. Zazwyczaj rozumowanie jest obarczone jakąś niepewnością, która zazwyczaj wynika z niedoskonałości wiedzy jaką posiada człowiek. Niedoskonałość owa niestety nie zwalnia z konieczności podejmowania decyzji, wyciągania wniosków, wydawania ekspertyz itp. Człowiek zazwyczaj sobie radzi z tego rodzaju trudnościami posiłkując się jakąś, często subiektywną i obarczoną dużym błędem, formą oceny niepewności, prawdopodobieństwa, przypuszczeń itp. Z jednej strony niepewne mogą przesłanki decyzji, a z drugiej niepewnością i innymi rodzajami niedoskonałości obarczone mogą być zasady jakimi dana osoba kieruje się podczas podejmowania decyzji. Dodatkowym czynnikiem komplikującym problem modelowania niepewności wiedzy jest jej charakter, tzn. niedoskonałość wiedzy może być związana z różnymi czynnikami, np. z losowością, lukami w wiedzy, przypuszczeniami, brakiem precyzji a, co więcej, zazwyczaj różne niedoskonałości posiadanej wiedzy nakładają się na siebie, dodatkowo utrudniając proces rozumowania. Każda z nich ma inną naturę, wywodzi się z innego spojrzenia na problem i wymagać może innego formalizmu do jej reprezentacji.

Chcąc realizować systemy ekspertowe, które chociażby częściowo, zbliżone były w swym działaniu do ludzkiego eksperta należałoby w jakiś sposób móc owe różne rodzaje niedoskonałości wiedzy zapisać w bazie wiedzy, gdyż paradoksalnie, im lepiej niedoskonałości owe będą tam reprezentowane tym

lepsze i dokładniejsze będą wnioski.

Opracowano szereg teorii reprezentowania niepewności w systemach ekspertowych. Większość źródeł literaturowych [11], [9], [14] podaje tu kilka podstawowych sposobów reprezentacji wiedzy niepewnej, niepełnej, nieprecyzyjnej itp.:
— Współczynniki pewności,
— Bayesowskie sieci przekonań,
— Zbiory rozmyte,
— Zbiory przybliżone,
— Inne (teoria Dampstera–Shafera, teoria możliwości).
Pierwszą próbą implementacji wnioskowania z niepełną i niepewną informacją był system wnioskowania ze współczynnikami pewności wykorzystany w systemie MYCIN [5]. Idea współczynników pewności leży w zaszyciu poziomu niepewności w wielkość współczynnika przywiązanego do określonego atrybutu. Współczynnik pewności może przyjąć wielkość z pewnego zakresu (zazwyczaj [–1,1]) [5], [26], [16]. Sposób reprezentacji wiedzy za pomocą współczynników pewności został potem poddany dość mocnej krytyce m.in. ze względu na niezgodność z logiką arystotelesowską i wpływ kolejności rozpatrywania reguł na wielkość współczynnika pewności wniosku [26].
Stanfordzka algebra współczynników pewności wykorzystana w systemie MYCIN została zmodyfikowana w systemach RMSE [16], [26]. Modyfikacje te doprowadziły m.in. do zgodności z logiką arystotelesowską, oraz wykluczyły wpływ kolejności rozpatrywania reguł na wielkość wynikowego współczynnika pewności [9], [26].
W literaturze [11] podkreślono przewagę systemów ze współczynnikami pewności w stosunku do systemów opartych na aparacie probabilistycznym, w medycynie, oraz ogólnie w zastosowaniach diagnostycznych. Problematyka oceny wniosków kredytowych w rzeczywistości jest również zadaniem o charakterze diagnostycznym, co potwierdza przydatność tego sposobu reprezentacji wiedzy.

Systemy bazujące na probabilistyce wykorzystują rozbudowany aparat matematyczny, pozwalający wyznaczać precyzyjnie prawdopodobieństwo, ale z drugiej strony wymagający trudnych do zdobycia, a czasem nawet i subiektywnego oszacowania danych.

Inną techniką ujęcia niepełności i niepewności danych w systemie ekspertowym opisaną m.in. w [11], [9], [27] jest wykorzystanie zbiorów rozmytych. W tym przypadku otrzymany z systemu rezultat, podobnie jak w przypadku systemów wykorzystujących współczynniki pewności, nie ma nic wspólnego z prawdopodobieństwem. Problemem w takich systemach jest konieczność

zdefiniowania funkcji przynależności, również, w wielu dziedzinach, bardzo trudnej do określenia.

Kolejną teorią, którą można wykorzystać do reprezentacji wiedzy niepewnej w systemie ekspertowym jest teoria Dempstera–Shafera [4]. Sieci bayesowskie mogą być traktowane jako podzbiór tej teorii. Idea teorii D.S. leży w tym, że określone wnioski zazwyczaj bazują na zbiorze hipotez, gdzie klasyczna teoria prawdopodobieństwa pozwala wyznaczyć prawdopodobieństwo dla jednej hipotezy, a nie dla ich zbioru. Teoria D.S. bazuje na lepszych podstawach matematycznych niż algebra współczynników pewności jednakże jest znacznie bardziej skomplikowana, ma większe wymagania dotyczące danych wejściowych i jest znacznie bardziej skomplikowana obliczeniowo.

Inne podejście do reprezentacji niedoskonałej wiedzy związane jest z wykorzystaniem teorii zbiorów przybliżonych, w której zbiór jest traktowany jako para zbiorów: przybliżenia górnego i dolnego.
Nie wszystkie z tych metod są powszechnie stosowane: niektóre wykorzystuje się częściej inne rzadziej. Generalnie w użyciu częściej spotyka się współczynniki pewności, bayesowskie sieci przekonań i zbiory rozmyte niż implementacje teorii Dempstera-Schafera, czy zbiory przybliżone, które ciągle częściej pojawiają się jako temat rozważań naukowych niż w praktycznych zastosowaniach.

Warto jednak wspomnieć, że pomimo dobrego odwzorowania charakteru wiedzy i jej niedoskonałości metody reprezentacji wiedzy niepewnej i niepełnej generalnie rzadko znajdują zastosowanie w praktyce. Przyczyna tego leży przede wszystkim w skomplikowanej reprezentacji tego rodzaju wiedzy, często znacznie odbiegającej od „przyjaznych" reguł, ale także w trudności w zdobyciu wszystkich potrzebnych do obliczeń danych, bądź problemów z kumulacją niepewności dla mocno zagnieżdżonej wiedzy. Nie zmienia to jednak faktu, że warto jest się im przyjrzeć, gdyż w pewnych sytuacjach mogą być one bardzo użyteczne.

Ogólnie rzecz biorąc w większości przypadków problem reprezentacji niepewności, rozmytości, braku wiedzy sprowadza się do przypisania pewnych liczbowych współczynników będących miarą niedoskonałości. Interpretacja oraz sposób liczenia i operacje na tych współczynnikach różnią się oczywiście, przez co każdy z nich reprezentuje innych rodzaj niedoskonałości wiedzy.

W dalszych podrozdziałach niniejszej książki zostaną przedstawione najpopularniejsze sposoby reprezentacji niedoskonałej wiedzy. Nie wszystkie z

nich będą przedstawione tak samo obszernie: część z nich czytelnik już prawdopodobnie poznał wcześniej a część ma ciągle jeszcze charakter badawczy i jest jeszcze bardzo daleka od powszechnego wykorzystania w praktyce.

3.5.1. Współczynniki pewności

Współczynniki pewności były prawdopodobnie pierwszą próbą wyrażenia niedoskonałości wiedzy w systemie ekspertowym. Po raz pierwszy były wykorzystane w systemie MYCIN [5]. Owa pierwsza implementacja była poddana wielokrotnie krytyce (m.in. [22]). Krytykowano przede wszystkim niezgodność z logiką arystotelesowską, niezbyt dobre oparcie w formalizmach rachunku prawdopodobieństwa i inne rzeczy, zainteresowanym historią sporów można polecić źródła ([5], [22]). W dzisiejszych czasach formalizm stanfordzkiej algebry współczynników pewności (bo tak był nazywany) jest traktowany już jako historia ale bardzo ciekawa okazała się jego rozbudowa proponowana przez prof. Niederlińskiego m.in w [16]. Plusem tej metody m.in. jest jej stosunkowa prostota oraz rozwiązanie problemu logiki arystotelesowskiej, która jest podzbiorem algebry współczynników pewności.

Metoda opiera się na klasycznych regułach typu „jeżeli... to...", o ile jednak w zwykłych regułach warunki i wnioski są zazwyczaj zmiennymi logicznymi, czyli przyjmują wartości prawda-fałsz, to w przypadku systemu niepewnego każdy warunek (i wniosek) ma współczynnik pewności (oznaczany jako CF), który przyjmuje wartość z zakresu $< -1; 1 >$. Pojawia się w takiej sytuacji oczywiste i bardzo ważne pytanie o definicję takiego współczynnika. W [16] definiuje się go jako subiektywną miarę przekonania o prawdziwości danego faktu, w [26] jest definiowany jako miara przekonania twórcy bazy o prawdziwości warunku. W obu tych definicjach istota współczynnika opiera się na ocenie prawdziwości danego faktu w bazie wiedzy. O ile w klasycznych systemach były dwa możliwe stany: prawda i nieprawda, to w systemach bazujących na współczynnikach pewności poziom prawdziwości i fałszu wyraża współczynnik pewności. Jeśli współczynnik pewności dla danego warunku (bądź wniosku) będzie wynosił:

— CF= -1 oznacza to, że dany warunek jest z całą pewnością nie jest prawdą,
— CF= -0.5 oznacza to, że dany warunek jest raczej fałszem,
— CF= 0 oznacza to, że warunek jest nieokreślony,
— CF= 0.5 oznacza to, że warunek jest raczej prawdą,
— CF= 1 oznacza to, że warunek jest na pewno prawdą.

Możliwe są oczywiście wszelkie stany pośrednie, czyli współczynnik pewności CF dla danego warunku A może wynieść np. $CF_A = 0.541$. Pojawia się w tym momencie bardzo zasadne pytanie o źródło takiego współczynnika, tzn.

jak zmierzyć pewność jakiegoś warunku? Ponieważ trudno jest zdefiniować jakąś jedną, precyzyjną i spójną metodę należy przyjąć, że współczynnik pewności jest po prostu szacowany przez eksperta podczas deklaracji faktów (w trakcie wnioskowania jest wyznaczany przez mechanizm wnioskujący). W sytuacji, gdy jest taka możliwość, poziom oceny prawdziwości danego faktu może wynikać z uśredniania kilku ocen dokonanych przez ekspertów, lecz generalnie trzeba się pogodzić z tym, że jest to wartość pochodząca z subiektywnej miary, przez co niedokładna i nieprecyzyjna. W związku z tym przykładowa wartość zadeklarowanego współczynnika pewności podana wyżej jest zadeklarowana ze zbyt wielką dokładnością. W praktycznych realizacjach system zaokrągla wartości współczynnika do dwóch miejsc po przecinku. Owa subiektywność oceny jest też traktowana jako główna wada metody: brak formalizmu matematycznego wyznaczania tego rodzaju wartości i precyzyjnej jej definicji powoduje pewne trudności w interpretacji współczynników pewności i całego procesu wnioskowania. Z drugiej strony patrząc, człowiek podczas rozumowania zazwyczaj posługuje się takim, nieprecyzyjnym i niedokładnie sformalizowanym, pojęciem pewności. Co więcej, posługuje się nim sprawnie i skutecznie.

Nie tylko warunki i wnioski mogą być niepewne. Niejednokrotnie także reguły, którymi posługują się ludzie nie mogą być traktowane jako absolutne pewniki. Każdy na pewno spotkał się z tym np. w trakcie wizyt u lekarza, który diagnozował, że mając takie a nie inne objawy pacjent najprawdopodobniej choruje na taką chorobę, nie może jednak wykluczyć, że jest coś innego. Można to zilustrować przykładem, wg którego mając gorączkę i dreszcze najprawdopodobniej jest się chorym na grypę ale nie można też wykluczyć malarii, której objawy są podobne (proszę przykładu nie traktować zbyt dosłownie: tak na prawdę autor nie ma wielkich doświadczeń z malarią). Z powyższego przykładu można wywnioskować, że po pierwsze żaden wniosek nie jest do końca pewny, po drugie pewność pierwszego (z grypą) jest znacznie wyższa niż drugiego. W związku z tym można przyjąć, że reguły również są niepewne i również im przypisać współczynnik pewności. Współczynnik pewności reguły za [16] zdefiniować jako subiektywną miarę pewności reguły a za [26] jako liczbę z przedziału $< -1; 1 >$, będąca miarą przekonania twórcy bazy o prawdziwości reguły. Taka definicja znów prowokuje pytanie o źródło tego współczynnika. Podobnie jak w przypadku współczynnika pewności warunku jego wartość może być po prostu szacowana przez ekspertów, bądź, jeśli jest dostęp do odpowiednich danych, jego wartość może pochodzić np. z analiz częstości spełniania tego rodzaju reguły w danych z przeszłości. W przypadku tego ostatniego sposobu należy wziąć pod uwagę również wiarygodność danych do analizy oraz ich ewentualne starzenie się: wykorzystanie tego rodzaju wyników np. przy ocenie

zdolności kredytowej klientów banku może być niebezpieczne, ze względu na to, że wymagania dotyczące dobrego kredytobiorcy mogły się w ciągu tego okresu znacząco zmienić. Nic nie stoi np. na przeszkodzie wykorzystać tego rodzaju metodę przy ustalania wartości współczynników pewności reguł w np. systemach medycznych.

Wartość współczynnika pewności reguły jest liczbą z zakresu $< -1; 1 >$ i interpretuje się ją tak [26]:

— CF_{reg} =1 oznacza, że wniosek jest prawdziwy dokładnie w takim stopniu, w jakim jest prawdziwa lista warunków; innymi słowy – lista warunków wspiera całkowicie pewność wniosku swoją pewnością;
— CF_{reg} =0.5 oznacza, że wniosek jest o połowę mniej prawdziwy aniżeli lista warunków; innymi słowy – lista warunków wspiera tylko w połowie pewność wniosku swoją pewnością;
— CF_{reg} =0 oznacza, że wniosek ma pewność niezależną od pewności listy warunków; innymi słowy – pewność list warunków nie ma wpływu na pewność wniosku.
— CF_{reg} = – 0.5 oznacza, że zanegowany wniosek ma pewność o połowę mniejszą aniżeli pewność listy warunków; innymi słowy – lista warunków osłabia w połowie pewność wniosku swoją pewnością;
— CF_{reg} = – 1 oznacza, że wniosek jest nieprawdziwy dokładnie w tym stopniu, w jakim jest prawdziwa lista warunków; innymi słowy – lista warunków wspiera całkowicie pewność zanegowanego wniosku swoją pewnością.

Współczynnik pewności reguły może być także interpretowany jako swego rodzaju wzmocnienie reguły.

3.5.1.1. Algebra współczynników pewności

Mając już zdefiniowane współczynniki pewności można przyjrzeć się temu jak są one przetwarzane w trakcie wnioskowania. System niepewny bazujący na współczynnikach pewności opiera się na regułach podobnych do tych w klasycznym ekspertowym. Reguła w systemie ekspertowym składa się z części warunkowej, czyli listy warunków (w analizowanym przypadku warunki z listy połączone są spójnikiem i) oraz wniosku. W klasycznej regule prawdziwość listy warunków implikowała prawdziwość wniosku. W przypadku systemu niepewnego pewność warunków implikuje pewność wniosku. Współczynnik pewności wniosku pojedynczej reguły wyznacza się z wzoru:

$$CF_{wniosku} = CF_{listywarunkow} \cdot CF_{reguly} \tag{3.1}$$

Współczynnik pewności wniosku zatem może być tylko tak pewny jak pewna jest reguła. Współczynnik pewności listy warunków wyznacza się korzysta-

jąc ze wzoru:

$$CF_{listawarunkow} = min(CF_{warunek1}, CF_{warunek2}, \ldots) \tag{3.2}$$

Wynika z niego,że lista warunków jest tak pewna jak pewny jest jej najmniej pewny warunek (warto tu pamiętać, że warunki są połączone spójnikiem *i*, gdyby był współczynnik *lub* wtedy liczone powinno być to w inny sposób). Przypomina to łańcuch, który jest tak mocny jak mocne jest jego najsłabsze ogniwo. W przypadku, gdy wśród listy warunków są warunki, które z jakąś pewnością są nieprawdą (mają ujemny współczynnik pewności), wtedy wniosek również będzie z jakąś pewnością nieprawdą

Odnosząc reguły niepewne do klasycznego systemu ekspertowego, można zauważyć, że gdyby pozwolić na przypisanie tylko dwóch wartości: 1 i -1, wtedy takie reguły działałyby tak samo jak pojedyncze reguły systemu klasycznego. Niestety wszystko to komplikuje się znacznie w sytuacji, w której jest kilka reguł o takim samym wniosku. W klasycznym systemie wystarczyło, że tylko jedna z tych reguł była spełniona (warunki były prawdziwe) i wniosek był również spełniony. W przypadku możliwości oceny niepewności wniosku istnienie innych reguł i poziom ich spełnienia może wpływać na pewność wniosku końcowego, należy się wobec tego najpierw zastanowić nad naturą warunków reguł, tzn.: W niektórych sytuacjach pewność wniosku powinna się w jakiś sposób kumulować (wg kilku reguł z jakąś pewnością wniosek jest prawdą, w związku z czym końcowy wniosek powinien być tym bardziej prawdą, czyli mieć jeszcze większy współczynnik pewności), w innych zaś owej kumulacji być nie powinno, czyli pewność wniosku końcowego powinna być ustalona przez jedną z listy reguł. Bazując na tym w [16] wprowadzono modyfikację algebry współczynników pewności [5] Modyfikacja owa polega na wprowadzeniu dwóch rodzajów reguł:
— reguły kumulatywne,
— reguły dysjunktywne.
Idea owego rozróżnienia opiera się na założeniu, że kilka reguł o tym samym wniosku może mieć listy warunków, które są od siebie niezależne, czyli wg. [16]:
dowolna wartość dowolnego współczynnika pewności z jednej listy warunków nie determinuje w żaden sposób wartość współczynnika pewności żadnego warunku z innej listy warunków,
lub zależne, czyli:
wartość współczynnika pewności z jednej listy warunków determinuje przynajmniej jeden współczynnik pewności drugiej listy warunków.

Zilustrować można to przykładami:

— Reguły kumulatywne:
`regula 1: dobra ocena kredytobiorcy jeżeli dobra ocena finansów`
`regula 2: dobra ocena kredytobiorcy jeżeli dobra reputacja kredytobiorcy`
Wnioskiem obu reguł jest dobra ocena kredytobiorcy, warunki każdej z reguł są od siebie niezależne, w związku z czym jeśli rzeczywiście jest z jakąś pewnością prawdą, że potencjalny kredytobiorca ma dobrą ocenę finansów i dobrą reputację, to pewność końcowa wniosku powinna być większa niż pewność wniosków cząstkowych z obu reguł (pewność się kumuluje).

— Reguły dysjunktywne:
`regula 1: dobra ocena finansów jeżeli bardzo dobra płynność finansowa`
`regula 2: dobra ocena finansów jeżeli dobra płynność finansowa`
Wnioskiem obu reguł jest dobra ocena finansów ale niewątpliwie reguła 1 będzie miała wyższy współczynnik pewności niż reguła 2. Ponieważ dobra i bardzo ocena płynności nie są względem siebie obojętne, to łączna pewność wniosku nie powinna wynikać z ich kumulacji.

Jak, w takim razie, powinny być wyznaczane współczynniki pewności wniosków końcowych dla grup reguł o takim samym wniosku?

Reguły kumulatywne Sposób wyznaczania opisany będzie na przykładzie dwóch prostych reguł o takim samym wniosku i jednym warunku:
`regula 1: Wniosek jeżeli A (`CF_{reg1}`)`
`regula 2: Wniosek jeżeli B (`CF_{reg2}`)`
Powyższe dwie reguły mają zadeklarowane swoje współczynniki pewności: CF_{reg1}, CF_{reg2}, warunki reguł również mają swoje współczynniki (zadeklarowane, bądź wyznaczone z innych reguł): CF_A i CF_B. Współczynniki pewności wniosków ($\mathrm{CF}_{wniosek1}$, $\mathrm{CF}_{wniosek1}$) z poszczególnych reguł wyznacza się z podanego wyżej wzoru, a celem jest wyznaczenie łącznego współczynnika pewności dla wniosku „wniosek” ($\mathrm{CF}_{wniosek}$).

Dla reguł kumulatywnych listy warunków muszą być od siebie niezależne, czyli współczynnik pewności warunku A nie może w żaden wpływać na współczynnik pewności warunku B i vice versa.

Wyznaczenie współczynnika pewności dla wspólnego wniosku dwóch reguł kumulatywnych na drodze działania zwanego kumulowaniem współczyn-

ników pewności. Kumulowanie odbywa się następująco:
Jeżeli:
$\mathrm{CF}_{wniosek1} \geq 0$ i $\mathrm{CF}_{wniosek2} \geq 0$
to wypadkowy $\mathrm{CF}_{wniosek}$ będzie równy:

$$CF_{wniosek} = CF_{wniosek1} + CF_{wniosek2} – CF_{wniosek1} \bullet CF_{wniosek2} \tag{3.3}$$

jeżeli:
$\mathrm{CF}_{wniosek1} < 0$ i $\mathrm{CF}_{wniosek2} < 0$
to wypadkowy $\mathrm{CF}_{wniosek}$ będzie równy:

$$CF_{wniosek} = CF_{wniosek1} + CF_{wniosek2} + CF_{wniosek1} \bullet CF_{wniosek2} \tag{3.4}$$

Jeżeli:
$\mathrm{CF}_{wniosek1} \geq 0$ i $\mathrm{CF}_{wniosek2} < 0$ lub $\mathrm{CF}_{wniosek2} \geq 0$ i $\mathrm{CF}_{wniosek1} < 0$
to wypadkowy $\mathrm{CF}_{wniosek}$ będzie równy:

$$CF_{wniosek} = \frac{CF_{wniosek} + CF_{wniosek}}{1 - min(|CF_{wniosek} + CF_{wniosek}|)} \tag{3.5}$$

Czyli:

Każdy dodatkowy wniosek z dodatnią wartością CF dla reguły kumulatywnej jest dodatkowym czynnikiem zwiększającym pewność zaistnienia tego wniosku. Pewność ta nie może jednak nigdy być większą od wartości CF = 1. Każdy dodatkowy wniosek z ujemną wartością CF dla reguły kumulatywnej jest dodatkowym czynnikiem zmniejszającym pewność zaistnienia tego wniosku. Pewność ta nie może jednak nigdy być mniejszą od wartości CF= – 1.

Przedstawione działanie uzasadniają jeszcze jedną interpretację współczynników pewności w przypadku reguł kumulatywnych: można uważać je za „współczynniki wagi", określające względny wpływ warunku A i warunku B na wspólny wniosek.
W przypadku większej liczby reguł kumulatywnych o współczynnikach pewności wniosku z tym samym znakiem korzysta się z wymienionej zasady najpierw dla dwóch reguł, otrzymany współczynnik pewności kumuluje się z współczynnikiem pewności wniosku dla trzeciej reguły, itd. W przypadku większej liczby reguł kumulatywnych o współczynnikach pewności wniosku z różnymi znakami postępuje się następująco:
— wyznacza się wypadkowy współczynnik pewności wniosku dla reguł, dla których jest on dodatni;
— wyznacza się wypadkowy współczynnik pewności wniosku dla reguł, dla których jest on ujemny;

— kumuluje się współczynnik pewności dodatni i ujemny.

Reguły dysjunktywne Wyznaczenie współczynnika pewności dla wspólnego wniosku reguł dysjunktywnych odbywa się zgodnie z następującymi zasadami:
Reguły dysjunktywne o ujemnych współczynnikach pewności, dla których w trakcie wnioskowania współczynnik pewności listy warunków przyjmuje wartości ujemne, są w dalszym ciągu wnioskowania pomijane. Zasadę tę można prosto uzasadnić na przykładzie reguł dysjunktywnych:
`regula 1: dobra ocena finansów jeżeli bardzo dobra płynność finansowa (CF`$_{reg1} = 0,8$)
`regula 2: dobra ocena finansów jeżeli brak płynności finansowej (CF`$_{reg2} = -1$)
Jeżeli CF(brak płynności finansowej)= – 1, CF(dobra płynność finansowa)=0,8, to CF(dobra ocena finansów) musi być wyznaczone przez regułę 1 jako równe 0.64, a nie przez regułę 2 jako równe 1. Innymi słowy: reguła dysjunktywna o ujemnym współczynniku pewności nie może być stosowana do generowania dodatniego współczynnika pewności wniosku, lecz jedynie do generowania ujemnego współczynnika pewności wniosku. Będzie to nosiło nazwę zasady równoczesnej nieujemności (CF reguły i CF warunków reguł dysjunktywnych).

Jeżeli dla reguł dysjunktywnych, które nie zostały pominięte w wyniku zastosowania zasady równoczesnej nieujemności, współczynniki pewności list warunków są różnych znaków, należy pominąć reguły dysjunktywne o ujemnych wartościach współczynników pewności. Zasadę tę można również prosto uzasadnić na powyższym przykładzie:

Jeżeli CF(brak płynności finansowej) = 0,6, CF(dobra płynność finansowa)= – 1, to CF(dobra ocena finansów) musi być wyznaczone przez regułę 4 jako równe - 0.6, a nie przez regułę 1 jako równe –1.
Można to uzyskać pomijając reguły dysjunktywne o ujemnych współczynnikach pewności list warunków.

Jeżeli dla reguł dysjunktywnych, które nie zostały pominięte w wyniku zastosowania zasady równoczesnej nieujemności, współczynniki pewności list warunków są jednakowych znaków, to wyznacza się wypadkowy współczynnik pewności wniosku jako największy spośród współczynników generowanych przez wymienione reguły.

Uwagi na temat reguł kumulatywnych i dysjunktywnych Warto zwrócić uwagę na to, że grupa reguł o takim samym wniosku nie może być w części kumulatywna a w części dysjunktywna. Chcąc wyrazić tego typu zależności można użyć atrybutów pośrednich, łącząc warunki dysjunktywne z jednym wnioskiem a ten kumulatywnie z resztą warunków kumulatywnych (dokładniej jest to opisane w [26]).
Powyższy model wnioskowania niepewnego z wykorzystaniem podziału na reguły kumulatywne i dysjunktywne z warunkami zależnymi i niezależnymi jest oczywiście pewnym uproszczeniem (dość dużym) rzeczywistości, gdyż czasem trudno jest wykluczyć pewne powiązanie między warunkami zdałoby się zupełnie niezależnymi (np. czy dobra reputacja kredytobiorcy na pewno nie wiąże się z jego dobrą pozycją na rynku?) i na odwrót. Pomimo tego jest to rozwiązanie użyteczne i zdecydowanie bliższe temu jak rozumuje człowiek, niż klasyczne systemy ekspertowe.
Jedną z cech stanfordzkiej algebry współczynników pewności zastosowanej w systemie MYCIN, była niezgodność z logiką arystotelesowską, inaczej rzecz biorąc nie dało się w tym systemie zrealizować wnioskowania takiego jak dla zwykłego systemu ekspertowego (bez współczynników pewności). Model algebry współczynników pewności wprowadzający rozróżnienie między regułami kumulatywnymi i dysjunktywnymi rozwiązywał ten problem. W takim systemie wystarczy wszystkie reguły oznaczyć jako dysjunktywne z CF=1 oraz przypisywać wszystkim warunkom dopytywalnym (tym, których CF nie można wyznaczyć z innych reguł) CF=1 lub CF=-1 (reprezentacja dwóch stanów logicznych: prawdy i fałszu).

3.5.1.2. Warunki wzajemnie wykluczające się

W klasycznych systemach ekspertowych część warunków mogła się wzajemnie wykluczać, tzn. jeśli jeden z warunków z listy był prawdą, to reszta na pewną była fałszem. W przypadku systemów niepewnych tego rodzaju ograniczenie również może zachodzić (jeden z warunków ma CF=1, reszta ma CF=-1) ale także mogą się wykluczać niepewnie, tzn. jeden może być z jakąś pewnością prawdą ($CF > 0$) a reszta z jakąś pewnością nieprawdą ($CF < 0$).

3.5.1.3. Nadmiarowości i sprzeczności

Uważny czytelnik niewątpliwie w tym momencie zada pytanie: „a jak wygląda kwestia anomalii: nadmiarowości i sprzeczności w niepewnej bazie wiedzy?". Jest to pytanie niewątpliwie słuszne i ważne w kontekście praktycznej realizacji niepewnych systemów ekspertowych. Generalnie sprzeczności dla systemu niepewnego są takie same jak dla systemu klasycznego, fakt dodania współczynnika pewności nie zmienia ich natury. Nieco inaczej jest natomiast w przypadku nadmiarowości, gdzie w sytuacji dwóch (lub

więcej) subsumujących się dysjunktywnych reguł: np.:
reguła 1: A jeżeli B (CF_{reg1})
reguła 2: A jeżeli B, C (CF_{reg2})
Nie musi to być nadmiarowość, gdyż reguła 2 może być uszczegółowieniem, czy też ostrzejszym warunkiem, dającym większą pewność wniosku, np.:
reguła 1: dostanę awans jeżeli znam język angielski (CF_{reg1}=0,4)
reguła 2: dostanę awans jeżeli znam język angielski, osiągnąłem dobre wyniki (CF_{reg2}=0,9)
Z pierwszej reguły wniosek „dostanę awans" jest znacznie mniej pewny, niż z drugiej, dlatego, jeśli jest w firmie ktoś, kto co prawda mówi po angielsku ale dotychczas nie wykazał się niczym szczególnym, to oczywiście ma szanse na awans ale znacznie większe szanse ma ktoś kto spełnił oba warunki.
Podobna sytuacja jest dla przypadku nadmiarowości dla dwóch wzajemnie wykluczających się warunków np.:
reguła 1: dostanę awans jeżeli nie znam języka angielskiego, osiągnąłem dobre wyniki (CF_{reg1}=0,4)
reguła 2: dostanę awans jeżeli znam język angielski, osiągnąłem dobre wyniki (CF_{reg2}=0,9)
W takim przypadku znów spełnienie warunków pierwszej reguły daje jakąś (niewielką) pewność, że dana osoba dostanie awans, natomiast spełnienie drugiej reguły (razem nie mogą być spełnione ze względy na wykluczający charakter warunków „znam język angielski" i „nie znam języka angielskiego") daje znacznie większą pewność awansu.
Powyższe przykłady pokazują, że wprowadzenie współczynników pewności powoduje, że w niektórych sytuacjach coś, co w klasycznym systemie jest nadmiarowością w systemie niepewnym nadmiarowością być nie musi (choć czasem może).

3.5.1.4. Problemy interpretacyjne i pułapki współczynników pewności

Algebra współczynników pewności jest dość prosta obliczeniowo, dodatkowo, oparcie na łatwych do zrozumienia regułach powoduje, że są one stosunkowo „przyjazne dla użytkownika" tzn. nie są trudne do zrozumienia nawet dla laika, co ma duże znaczenie podczas praktycznej realizacji takiego systemu, gdyż eksperci dziedzinowi, których wiedza jest implementowana w systemie niekoniecznie są biegli w informatyce w ogólności a w systemach ekspertowych w szczególności.

Niestety praktyczna realizacja systemu nie jest wolna od problemów. Pierwszy z nich wiąże się tzw. kumulacją niepewności. Owa kumulacja niepewności polega na tym, że w mocno zagnieżdżonej niepewnej bazie wiedzy pewność wniosku końcowego nawet dla idealnie pewnego warunku może

spaść do pomijalnie małej wartości. Zilustrować można to prostym przykładem bazy wiedzy:

reguła 1: A jeżeli B (CF_{reg1}=0,8)
reguła 2: B jeżeli C (CF_{reg2}=0,5)
reguła 3: C jeżeli D (CF_{reg3}=0,6)
reguła 4: D jeżeli E (CF_{reg4}=0,4)
reguła 5: E jeżeli F (CF_{reg5}=0,9)
reguła 6: F jeżeli G (CF_{reg6})=0,5

Zakładając, że warunek G jest prawdą i jest całkowicie pewny, czyli CF(G)=1, to wnioski poszczególnych reguł będą wyglądać tak:
CF(F)=0,5; CF(E)=0,45; CF(D)=0,18; CF(C)=0,108; CF(B)=0,054; CF(A)=0,0432

Takie wyniki oznaczają, że pewność wniosku A wynosi zaledwie 0,0432, czyli nawet całkowicie pewny warunek nie daje żadnej znaczącej informacji na temat pewności wniosku końcowego. Nie musi to znaczyć, że wnioskowanie jest niepoprawne po prostu nastąpiła kumulacja niepewności: wniosek reguły niepewnej jest niepewny, jeśli ten niepewny wniosek będzie warunkiem kolejnej niepewnej reguły, jej wniosek będzie jeszcze bardziej niepewny, gdy ten wniosek będzie warunkiem jeszcze jednej reguły itd. Co w takim razie można z tym zrobić? po pierwsze nie ma sensu dawać reguł o bardzo małej pewności: ich wpływ na wniosek jest pomijalnie mały, po drugie bazy wiedzy niepewne nie powinny mieć aż tak mocnych zagnieżdżeń: proces wnioskowania w oparciu o niepewne warunki i reguły w takiej sytuacji zawsze przy pewnej liczbie reguł staje się bezcelowy, zilustrować to można kolejnym prostym przykładem:

Pewna osoba o imieniu Zenek siedząc przed telewizorem poczuła drapanie w gardle, w związku z czym donośnie kichnęła. Owo kichnięcie uruchomiło pewien proces niepewnego wnioskowania. Zenek myśli sobie tak: kichnąłem, w związku z czym może jestem chory? może chory na grypę? jeśli jestem chory na grypę to pewnie już zarażam, skoro dziś widziałem się ze swoją przyjaciółką Zosią, to może ją zaraziłem? a może grypa na którą choruję jest grypą odzwierzęcą? mam kota, więc może to kocia grypa? może to jest jakaś szczególnie niebezpieczna odmiana tej choroby? Jeśli jest to poważna choroba, to pewnie położą mnie w szpitalu w izolatce, Jeśli Zosia zaraziła się ową niebezpieczną grypą ode mnie to pewno już niedługo zacznie chorować? Jak będzie poważnie chorować to będzie w musiała iść do szpitala? Może położą nas w tej samej izolatce?

Ów powyższy przykładowy proces wnioskowania Zenka wychodził od całkowicie pewnego warunku (kichnięcia) a skończył się wnioskiem praktycznie nieprawdopodobnym (wspólne chorowanie w jednej izolatce). Przykład ten demonstruje, że problem zagnieżdżania i kumulacji niepewności występuje także we wnioskowaniu ludzkim.

Problemu kumulacji niepewności nie da się całkowicie zlikwidować, dlatego podczas interpretacji wyników wnioskowania należy brać je pod uwagę i odnosić pewność otrzymanego wniosku do maksymalnej możliwej pewności jaką można uzyskać dla tego wniosku [26].
Wspomnianą już wcześniej wada współczynników pewności jest brak umocowania w konkretnych formalizmach matematycznych, bazuje ona na intuicyjnym rozumieniu niepewności nie precyzując tak na prawdę do końca czym ona jest i jak należy ją interpretować. Może to stwarzać zagrożenie polegające na myleniu pewności warunku z np. stopniem jego spełnienia.

3.5.1.5. Podsumowanie problematyki współczynników pewności

Podsumowując tematykę współczynników pewności warto zauważyć kilka charakterystycznych ich cech: po pierwsze mechanizm wnioskowania bazujący na współczynnikach pewności jest próbą odwzorowania intuicyjnego procesu rozumowania w sytuacji niepewności warunków i reguł. Po drugie współczynniki pewności bazują na stosunkowo prostej obliczeniowo i łatwej do zrozumienia algebrze, która nie wymaga dużej liczby trudnych do zdobycia lub oszacowania danych. Po trzecie współczynniki pewności można traktować jako rozszerzenie klasycznego systemu ekspertowego o dodatkowe informacje dotyczące pewności warunków, co może mieć niebagatelne znaczenie dla praktycznych zastosowań: jeśli urzędnik bankowy podejmuje decyzję o przyznaniu kredytu firmie, której system ekspertowy dał ocenę „dobry kredytobiorca" to woli zapewne wiedzieć, czy pewność oceny jest wysoka, czy niska, bo jeśli niska to może konieczne byłoby zażądanie lepszego zabezpieczenia kredytu?

3.5.2. Sieci bayesowskie

Reprezentacja niepewności z wykorzystaniem współczynników pewności była jedną z pierwszych wykorzystaną w praktyce. Niestety była ona również mocno krytykowana. Mechanizmem, który jest swego rodzaju odpowiedzią na krytykę współczynników pewności są tzw. sieci bayesowskie.

Sieci bayesowskie w przeciwieństwie do współczynników pewności opierają się na rachunku prawdopodobieństwa i jego wynik interpretowany jest jako prawdopodobieństwo, chociaż często są poważne problemy z wyznaczeniem prawdopodobieństw koniecznych do policzenia prawdopodobieństwa wyniku wnioskowania. Prawdopodobieństwo jest kategorią bardzo ściśle zdefiniowaną w matematyce i jest ono związane z częstością, w sytuacjach rzeczywistych, pomimo, że termin ten jest w powszechnym użyciu rzadko zdarza się, że jest on wykorzystywany poprawnie.
W bardzo wielu, być może nawet większości, sytuacji życiowych, niezwykle

trudno jest policzyć prawdopodobieństwo jakiegoś zdarzenia losowego, bo jak wyznaczyć prawdopodobieństwo zdania egzaminu? Najprościej można tak: „są dwie możliwości zdam albo nie zdam, w związku z tym prawdopodobieństwo wychodzi P(Z)=0,5", inna możliwość to: „skoro dotychczas na 8 egzaminów, które miałem, zdałem 7, to prawdopodobieństwo tego, że zdam P(Z)=0,875", jeszcze inaczej można tak: „jak na razie egzamin ten zdało 123 osoby na 145 podchodzących, w związku z czym prawdopodobieństwo, że zdam wyniesie P(Z)=0,85" itd. można mnożyć sposoby wyznaczania owego prawdopodobieństwa bazując na różnych przesłankach, nie wchodząc jednak w meritum sprawy, tzn. czy dany delikwent nauczył się na egzamin, czy też nie, bo to czy ktoś zda egzamin jest w przeważającej przecież mierze zależne właśnie od tego (choć oczywiście nie tylko: każdy student może podać przykłady, że umiał i nie zdał i na odwrót).
W praktyce często używa się subiektywnego prawdopodobieństwa opartego na szacunkach, w odróżnieniu od obiektywnego, opartego ściśle na faktach i statystyce [4]. Znajomość samego prawdopodobieństwa nie rozwiązuje jednak problemu rozumowania z prawdopodobieństwem, konieczna jest jakaś forma reprezentacji problemu zależności między losowymi zdarzeniami, np. jakie jest prawdopodobieństwo, że egzaminowany student zda w sytuacji, gdy się nauczył a jakie, że zda kiedy się nie nauczył?

Celem niniejszej pracy nie jest oczywiście wprowadzenie do rachunku prawdopodobieństwa, który każdy czytelnik niewątpliwie miał w szkole, warto jednak na początek przyjrzeć się samej jego definicji: generalnie prawdopodobieństwo definiuje się na kilka sposobów, najbardziej znana definicja określa je jako liczbę z zakresu $< 0; 1 >$ reprezentującą liczbę zdarzeń sprzyjającą jakiemuś zdarzeniu Z w stosunku do wszystkich możliwych zdarzeń. Niestety definicja ta nie dość, że zawiera błędy (zainteresowanym autor proponuje ich poszukać) to jest niestety, ze względu na trudności w uzyskaniu wiarygodnych wyników, mało użyteczna w wielu praktycznych zastosowaniach. W największym uproszczeniu, można prawdopodobieństwo traktować jako ocenę szansy, że zajdzie oczekiwane zdarzenie. Na użytek niniejszej pracy właśnie tak będzie ono interpretowane.

Gdyby próbować przedstawić pełny opis prawdopodobieństw jakiejś sytuacji z rzeczywistości składającej się z grupy N zdarzeń atomowych (np.: Maciek jest wysoki, Zosia jest prawnikiem, Józek jest żonaty), to konieczne było by przedstawienie 2^N różnych prawdopodobieństw odnoszących się do wszystkich możliwych układów „prawda – nieprawda" wszystkich zdarzeń (np. prawdopodobieństwo, że prawdą jest: Maciek jest wysoki, Zosia jest prawnikiem i nieprawdą, że Józek żonaty). Łączna suma tych prawdopodobieństw będzie równa oczywiście 1, co oznacza, że jedno z nich na pewno

będzie prawdą [4]. Oczywiście próba opisu jakiejkolwiek rzeczywistej sytuacji za pomocą tego rodzaju metody jest co najmniej trudna do realizacji praktycznej, ponieważ nie wiadomo skąd wziąć wszystkie te prawdopodobieństwa, brak w takiej sytuacji nawet jakichkolwiek podstaw (poza zupełną przypadkowością) do tego aby prawdopodobieństwa te chociażby intuicyjnie szacować. Tego rodzaju rozwiązanie jest więc zupełnie bezużyteczne w kontekście reprezentacji wiedzy i wspomagania procesu rozumowania.
Aby powyższy model uprościć można poczynić jedno, dość mocne, założenie niezależności wszystkich N zdarzeń. Wtedy prawdopodobieństwo jakiegoś układu tych zdarzeń będzie równe iloczynowi prawdopodobieństw zdarzeń cząstkowych. Niestety owo założenie jest zbyt optymistyczne: w zdecydowanej większości sytuacji zdarzenianie są niezależne, wpływają na siebie i aby możliwe było poprawne zamodelowanie procesu rozumowania z wykorzystaniem prawdopodobieństwa konieczna jest możliwość reprezentacji wpływu jednych zdarzeń na inne.

Znacznie lepszym pomysłem jest wykorzystanie prawdopodobieństwa warunkowego, które wyznacza się tak:

$$P(a|b) = \frac{P(a \wedge b)}{P(b)}$$

Korzystając z powyższego wzoru można wyznaczyć prawdopodobieństwo zdania egzaminu pod warunkiem nauczenia się i policzyć można go tak:

$$P(\text{zdam egzamin}|\text{nauczę się}) = \frac{P(\text{zdam egzamin} \wedge \text{nauczę się})}{P(\text{nauczę się})}$$

Problem korzystania ze wzoru na prawdopodobieństwo warunkowe znów wiąże się z dostępem do niektórych danych, szczególnie prawdopodobieństwa zajścia dwóch wydarzeń (P(zdam egzamin $\wedge$ nauczę się). W takiej sytuacji ratunkiem jest wzór bayesa, wg którego prawdopodobieństwo warunkowe można obliczyć tak:

$$P(a|b) = \frac{P(b|a) \cdot P(a)}{P(b)}$$

Przydatność tego wzoru dobrze ilustruje przykład medyczny: Chcąc obliczyć prawdopodobieństwo grypy mając objaw wysokiej gorączki. Ponieważ zdobycie prawdopodobieństwa współwystepowania tych dwóch zjawisk jest dość trudne to można posłużyć się wzorem bayesa: wyznaczenie prawdopodobieństw atomowych zdarzeń jest stosunkowo proste poprzez analizę np. częstości dotychczasowych chorób w kartotekach. Podobnie można zdobyć prawdopodobieństwo gorączki przy grypie (również dysponując danymi z

kartotek: liczba przypadków gorączki przy zdiagnozowanych przypadkach grypy). Prawdopodobieństwo grypy mając objaw wysokiej gorączki wynosi wtedy:

$$P(\text{grypa}|\text{wysoka gorączka}) = \frac{P(\text{wysoka gorączka}|\text{grypa}) \cdot P(\text{grypa})}{P(\text{wysoka gorączka})}$$

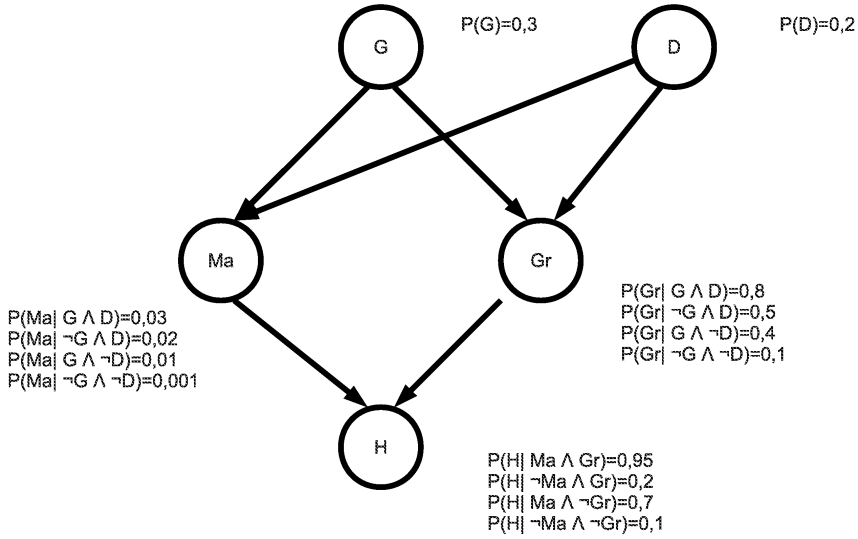
Wzór bayesa ma tą zaletę, że wymaga łatwiejszych do zdobycia danych, przez co ma szanse na zaaplikowanie do modelowania rzeczywistych procesów rozumowania.

Najpopularniejszym sposobem wykorzystania wzoru bayesa i generalnie rachunku prawdopodobieństwa w sztucznej inteligencji się tzw. sieci przekonań (bądź bayesowskie sieci przekonań).

Sieć przekonań jest acyklicznym grafem skierowanym, w którym węzły reprezentują zdarzenia losowe natomiast krawędzie reprezentują wpływ jednego zdarzenia na inne. Zamiast szacowania pełnego rozkładu prawdopodobieństw dla wszystkich układów wszystkich możliwych zdarzeń korzystając z sieci przekonań można wyznaczyć sobie prawdopodobieństwa poszczególnych zdarzeń. Zasada wyznaczania prawdopodobieństw opiera się na Zasada korzystania z sieci przekonań zostanie opisana na poniższym przykładzie:

Występowanie wysokiej gorączki i/lub dreszczy może oznaczać zdiagnozowanie grypy. Podobne objawy mogą też prowadzić do zdiagnozowania malarii. W niektórych przypadkach grypa może poważny charakter i prowadzić do konieczności hospitalizacji chorego. Podobnie malaria może mieć na tyle poważnych charakter, że będzie prowadzić do hospitalizacji chorego. Oczywiście prawdopodobieństwo tego, że chory na malarię będzie hospitalizowany jest odpowiednio większe od prawdopodobieństwa hospitalizacji. W sumie jest 5 zdarzeń:
— G - gorączka,
— D - dreszcze,
— Gr - grypa,
— Ma - malaria,
— H - hospitalizacja.
zależności między poszczególnymi można zdefiniować tak: Każde zdarzenie ma określone rozkłady prawdopodobieństw warunkowych dla wszystkich kombinacji prawda/fałsz wpływających na nie zdarzeń. Chcą na bazie powyższego przykładu wyznaczyć jakie jest prawdopodobieństwo hospitalizacji można je wyznaczyć, ze wzoru:

$$P(H) = P(H \mid Ma \wedge Gr) \cdot P(Ma) \cdot P(Gr) + P(H \mid Ma \wedge \neg Gr) \cdot P(Ma) \cdot P(\neg Gr) +$$



Rysunek 3.2. Przykładowa sieć przekonań.

$$P(H \mid \neg Ma \wedge Gr) \cdot P(\neg Ma) \cdot P(\ Gr) + P(H \mid \neg Ma \wedge \neg Gr) \cdot P(\neg Ma) \cdot P(\neg Gr)$$

Oczywiście konieczne tu prawdopodobieństwa grypy i malarii powinny być obliczone w podobny sposób, z uwzględnieniem prawdopodobieństw wpływu zdarzeń.

Chcąc wyznaczyć łączne prawdopodobieństwa różnych konkretnych zdarzeń np.: prawdopodobieństwo grypy, tego że pacjent jest hospitalizowany, miał gorączkę i nie miał dreszczy, czyli: P(Gr | H ∧ G ∧ ¬ D). Należy znów posłużyć się prawdopodobieństwem warunkowym sumując jednocześnie prawdopodobieństwa zdarzeń nie wymienionych w części warunkowej.

Generalnie prawdopodobieństwo P(Gr | H ∧ G ∧ ¬ D) wyznaczyć można ze wzoru:

$$P(Gr \mid H \wedge G \wedge \neg D) = \frac{P(Gr \wedge H \wedge G \wedge \neg D)}{P(H \wedge G \wedge \neg D)}$$

Prawdopodobieństwa koniunkcji wyznacza się jako sumy prawdopodobieństw warunkowych dla wszystkich możliwych kombinacji zdarzeń, czyli: $P(Gr \wedge H \wedge G \wedge \neg D) = P(Gr \wedge H \wedge G \wedge \neg D \wedge Ma) + P(Gr \wedge H \wedge G \wedge \neg D \wedge \neg Ma)$

Gdzie:

— $P(Gr \wedge H \wedge G \wedge\ (1 - P(D)) \wedge Ma) = P(Ma \mid G \wedge (1 - P(D))) \cdot P(G) \cdot (1 - P(D)) \cdot P(Gr \mid G \wedge (1 - P(D))) \cdot P(G) \cdot (1 - P(D)) \cdot P(H \mid Ma \wedge Gr)$

— $P(Gr \wedge H \wedge G \wedge\ (1 - P(D)) \wedge \neg Ma) = P(\neg Ma \mid G \wedge (1 - P(D))) \cdot P(G) \cdot (1 - P(D)) \cdot P(Gr \mid G \wedge (1 - P(D))) \cdot P(G) \cdot (1 - P(D)) \cdot P(H \mid \neg Ma \wedge Gr)$

Bazując na powyższym schemacie można wyznaczyć prawdopodobieństwa dowolnego układu zdarzeń.
Uważny czytelnik niewątpliwie zauważy pewną, wynikającą z daleko idącego uproszczenia, wadę powyższego rozwiązania, otóż zakłada się całkowitą niezależność zdarzeń nie połączonych ze sobą krawędziami. Wadą sieci przekonań jest także to, że są dość kłopotliwe przy dużej liczbie zdarzeń, często też bardzo trudno jest oszacować wszystkie potrzebne rozkłady zdarzeń, co mocno komplikuje praktyczne możliwości realizacji takich systemów.

Sieci przekonań pozwalają na oszacowanie prawdopodobieństw zdarzeń w sytuacji, gdy znana jest tylko część przesłanek, przez co pozwalają na wywnioskowanie prawdopodobieństw dla niepełnej wiedzy na temat niektórych warunków. Zaletą wnioskowania bayesowskiego jest również precyzyjna definicja i formalizacja mechanizmu. Jest ono najczęściej stosowane w sytuacjach, w których procesy wnioskowania nie są zbyt skomplikowane a konieczne jest szacowanie prawdopodobieństw zdarzeń przy niepełnych danych na temat poszczególnych zdarzeń. Mechanizm ten jest szczególnie często stosowany podczas różnego rodzaju ekspertyz medycznych.

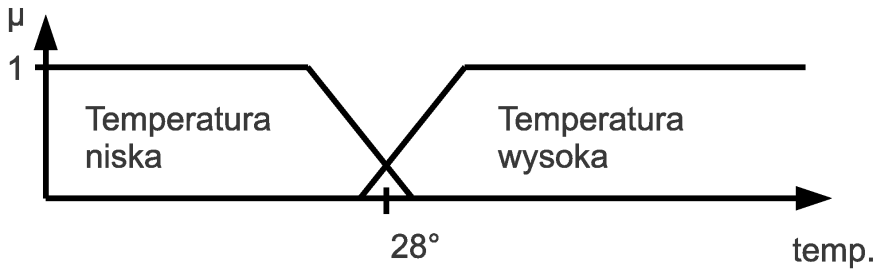
3.5.3. Zbiory rozmyte i logika rozmyta

Pojęcie zbioru rozmytego pojawiło się w pracy L. Zadeha [24] i zdefiniowany jest on tam jako uogólnienie zbioru klasycznego w postaci:

$$A = \{(x, \mu_A(x)) : x \in X, \mu_A(x) \in [0, 1]\}$$

gdzie x to element zbioru A, zaś $\mu_A(x)$ to funkcja przynależności, która wyraża stopień z jakim element x należy do zbioru A. Jeśli $\mu_A(x) = 0$ oznacza to, że element x nie należy do zbioru A, jeśli $\mu_A(x) = 1$ oznacza to, że element x całkowicie należy do zbioru, możliwe są także stany pośrednie, czyli mogą być elementy, które częściowo, bądź w jakimś stopniu należą do zbioru. Bez trudu można zauważyć, że w elementy klasycznych zbiorów mogą mieć tylko $\mu_A(x) = 0$ (gdy element nie należy do zbioru) lub $\mu_A(x) = 1$ (gdy element należy do zbioru), dzięki czemu zbiory rozmyte są uogólnieniem klasycznych zbiorów. Możliwość częściowej bądź niepełnej przynależności do zbioru pozwala na zamodelowanie rozumowania z wykorzystaniem powszechnych w naturalnym języku nieprecyzyjnych sformułowań. Trudno jest zdefiniować dokładne granice między pojęciami „niska temperatura" i „wysoka temperatura": nawet jeśli arbitralnie uzna się, że gorąco jest jeśli temperatura jest większa od 28 stopni celsjusza, to przecież nie ma wielkiej różnicy między 28,1 stopnia a 27,98 stopnia. Zbiory rozmyte pozwalają znacznie lepiej je opisać wykorzystując do tego częściową przynależność do zbioru. Na poniższym rysunku zademonstrowany jest przykład

dwóch zbiorów rozmytych: „Niska temperatura" i „Wysoka temperatura" na osi poziomej przedstawiona jest temperatura, na osi pionowej wartość funkcji przynależności do odpowiednich zbiorów. „Okolice" 28 stopni nie są



Rysunek 3.3. Przykładowe zbiory rozmyte.

do końca ani niską ani wysoką temperaturą, dlatego funkcja przynależności dla nich jest niższa niż dla „czystych" przypadków wysokiej i niskiej temperatury.

Czytelnik niewątpliwie już spotkał się (bądź spotka się niebawem) z tematyką zbiorów, logiki i wnioskowania rozmytego na innych przedmiotach. Jest ona również szeroko omówiona w wielu pozycjach literaturowych [27], [22].

3.5.4. Zbiory przybliżone

Inne ujęcie niedoskonałości wiedzy wykorzystywanej we wnioskowaniu zaproponował Zdzisław Pawlak [19]. Zbiory przybliżone bazują na odmiennym, niż w w poprzednich metodach podejściu: w zbiorach przybliżonych nie definiuje się żadnego współczynnika, czy też liczbowej miary stopnia „przybliżenia" zbioru. Koncepcja zbioru przybliżonego opiera się na tym, że zazwyczaj problem oceny tego, czy dany element przynależy do zbioru opiera się na jego cechach i po nich a właściwie po ich wartościach w trakcie procesu decyzyjnego rozpoznaje się, czy obiekt przynależy do zbioru, czy też nie. Wiąże się to z pewnymi konsekwencjami: przede wszystkim nie wszystkie atrybuty (bądź zbiory atrybutów) pozwalają zidentyfikować przynależność do zbioru.

W rzeczywistości trudno jest znaleźć dwa identyczne obiekty, prawie zawsze są jakieś cechy, którymi one się różnią. W wielu jednak sytuacjach możliwe jest pewne uproszczenie ich opisu i w konkretnych zastosowaniach mogą one być traktowane jako identyczne. Do ich definicji można wykorzy-

stać różne zbiory cech: niektóre z nich są wystarczające do pełnego zdefiniowania każdego obiektu, niektóre są nadmiarowe, niektóre zaś nie pozwalają na opisanie cech charakterystycznych pewnej grupy obiektów.
Podstawą rozważań w teorii zbiorów przybliżonych jest pewien zbiór obiektów U. Zbiór Q jest zbiorem cechy obiektów, a zbiór V jest zbiorem wartości tych cech. Funkcja f przyporządkowuje dla każdego obiektu wartości ze zbioru V cechom ze zbioru Q.

Bazując na powyższym, system informacyjny jest to czwórka:

$$SI =< U, Q, V, f >$$

Jeżeli zbiór cech Q podzieli się na dwie komplementarne grupy:
— atrybuty warunkowe: C,
— atrybuty decyzyjne: D.
to wtedy piątka:

$$DT =< U, C, D, V, f >$$

będzie nazywała się tablica decyzyjną i będzie ona reprezentowała zależność warunkową typu:

$$\textbf{jeżeli } \mathrm{C}_1 = \mathrm{V}_{c1v1} \textbf{ i } \mathrm{C}_2 = \mathrm{V}_{c2v1} \textbf{ wtedy } \mathrm{D}_1 = \mathrm{V}_{d1v1}$$

Jeżeli dla danego obiektu cecha $\mathrm{C}_1 = \mathrm{V}_{c1v1}$ i cecha $\mathrm{C}_2 = \mathrm{V}_{c2v1}$ to cecha decyzyjna D_1 będzie miała wartość $\mathrm{D}_1 = \mathrm{V}_{d1v1}$. Możliwa jest oczywiście większa liczba cech po stronie warunkowej jak i po stronie decyzyjnej [23].
Teoria zbiorów przybliżonych wprowadza szereg pojęć pozwalających na lepsze opisywanie analizowanych zbiorowości, poniżej zaprezentowano kilka z nich [23]:
— Relacja P-nierozróżnialności: Jeżeli dwa obiekty należące do U mają takie same wartości wszystkich cech należących do zbioru P ($\mathrm{P} \subseteq \mathrm{Q}$) to mówi się, że są P-nierozróżnialne.
— Klasa abstrakcji danej relacji P-nierozróżnialności jest to zbiór wszystkich obiektów będących w relacji P-nierozróżnialności.
— Dolna aproksymacja zbioru X: Zbiór obiektów, które na podstawie wartości cech P można z całą pewnością zakwalifikować do zbioru X nazywa się dolną aproksymacją zbioru X.
— Górna aproksymacja zbioru X: Zbiór obiektów, które na podstawie wartości cech P nie możemy jednoznacznie ocenić, że nie należą do zbioru X.
— Obszar brzegowy zbioru X definiuje się jako różnicę między jego górną i dolną aproksymacją względem zbioru cech P.
Powyższa lista pojęć nie jest oczywiście pełna, zainteresowani czytelnicy mogą zajrzeć tu [23] bądź tu [19].

Opisany powyżej skrótowo sposób reprezentacji braków w wiedzy za pomocą zbiorów przybliżonych jest narzędziem przydatnym podczas realizacji różnego rodzaju narzędzi klasyfikujących, pozwala też na wyrażenie niedoskonałości zdefiniowanych zasad i reguł.

3.5.5. Inne metody reprezentacji niedoskonałości wiedzy

Oprócz opisanych wyżej metod opracowano również szereg innych sposobów wyrażania niedoskonałości wiedzy, które stosunkowo rzadko znajdują praktyczne zastosowanie. Przyczyna tego leży często w zbyt skomplikowanym aparacie matematycznym, często nie przystającym do trudności problemu, bądź w trudnościach ze jego zrozumieniem wśród potencjalnych użytkowników systemu. Najpopularniejszą z tych metod jest teoria Dampstera-Schaffera [22], która bazując na rachunku prawdopodobieństwa odnosi się nie tylko do pojedynczych zdarzeń ale analizuje wszystkie możliwe podzbiory zdarzeń. Wprowadza pojęcia bazowego przyporządkowania prawdopodobieństwa (m), które nie musi być dane dla wszystkich elementów przestrzeni zdarzeń a jedynie na niektóre jej podzbiory. Teoria D-S definiuje warunki graniczne bazowego podporządkowania prawdopodobieństwa zakładając, że dla pustego zbioru zdarzeń m=0, a dla całej przestrzeni analizowanych zdarzeń m=1, zdefiniowane są również pewne sposoby wyznaczania bazowego przyporządkowania prawdopodobieństwa dla niepełnych i sprzecznych danych kierując się istniejącymi przesłankami.
Teoria D-S definiuje również pojęcie przekonania (suma m dla wszystkich zdarzeń z analizowanego zbioru) i wiarygodności (suma m dla części wspólnych analizowanych zbiorów zdarzeń).
Generalnie teoria D-S pozwala na realizację modelu, który rozróżnia prawdopodobieństwo zdarzeń od braku wiedzy na ich temat.

Inna metoda wyrażania niedoskonałości i niepełności wiedzy to teoria możliwości stworzona przez L. Zadeha, która bazuje na niuansach związanych z prawdopodobieństwem, wprowadzając pojęcia możliwości i konieczności.

Opisane wyżej sposoby wyrażania niuansów i niedoskonałości wiedzy stosunkowo rzadko znajdują zastosowanie w praktyce. Znacznie częściej są tematem prac naukowo badawczych niż praktycznych zastosowań biznesowych.

3.6. Ontologie

Ze względu na to iż ramy i sieci semantyczne w czystej formie jako reprezentacja wiedzy w systemach ekspertowych generalnie wyszły (bądź wychodzą) z użycia zastępowane przez ontologie wyrażone za pomocą np. logiki deskrypcyjnej autor proponuje czytelnikowi przyjrzeć się się właśnie ontologiom i najpopularniejszym sposobem ich reprezentacji.

Na wstępie warto zastanowić się czym jest owa tajemnicza ontologia. Termin ten wywodzi się z filozofii i jest to dział filozofii zajmujący się teorią bytu, badaniem charakteru i struktury rzeczywistości. Tematem jej badań jest struktura rzeczywistości: byty materialne, niematerialne, abstrakcyjne, próbuje odpowiedzieć na pytanie: czym jest byt, jak można byty scharakteryzować, czy i jakie są relacje między bytami itp. W informatyce pod tym terminem rozumie się pewien formalny opis pojęć określonej dziedziny – klasy (koncepty, pojęcia), role (właściwości, sloty) istniejące ograniczenia (fasety), relacje między bytami itp. Właśnie wspomniane wcześniej ramy i sieci semantyczne były jednymi z pierwszych próbami wyrażania opisu charakteru i struktury rzeczywistości.

3.6.1. Logika opisowa

Najpopularniejszym (choć nie jedynym) narzędziem do reprezentacji ontologii w systemach komputerowych jest logika opisowa (termin *description logic* jest również tłumaczony jako logika deskrypcyjna, bądź nie jest tłumaczony w ogóle). Logika deskrypcyjna jest podzbiorem logiki pierwszego rzędu i dziedziczy dzięki temu jej podstawowe cechy. Na język logiki opisowej składają się dwa podstawowe elementy:

— Tbox – terminologia ontologii, na którą składają się:
  — pojęcia,
  — role,
  — konstruktory.
— Abox – opis świata: instancje elementów ontologii.

W języku logiki opisowej można wyróżnić klasy (pojęcia) i role atomowe oraz konstruktory, które służą do definicji ról i pojęć złożonych. klasy atomowe to elementarne pojęcia, których nie można zdefiniować za pomocą innych pojęć i konstruktorów. W zbiorze pojęć atomowych oprócz pojęć deklarowanych przez twórcę ontologii wyróżnia się dwa podstawowe koncepty: $\top$ (top), który oznacza wszystko (klasa uniwersalna) i $\bot$ bottom, który oznacza nic – najniższy poziom ontologii, koncept, który nie ma wystąpień. Konstruktory języka logiki opisowej pozwalają na tworzenie złożonych klas i ról. Nieco

bardziej formalnie można elementy logiki deskrypcyjnej zdefiniować tak:

— klasa atomowa (pojęcie atomowe, koncept atomowy): A, B, C,
— Rola atomowa: R,
— Stała: a, b,
— $\sqcap$: część wspólna dwóch pojęć,
— $\exists$R.A: kwantyfikator egzystencjalny, zbiór takich obiektów, które chociaż raz powiązane rolą R z obiektem klasy A,
— $\forall$R.A: kwantyfikator ogólny: zbiór obiektów, których wszystkie powiązania rolą R dotyczą obiektów pojęcia (klasy) A,
— $\sqsupseteq$: zawieranie się,
— $\doteq$: równoważność,
— $\top$: koncept uniwersalny, czyli klasa: „coś”,
— $\bot$: koncept oznaczający „nic”.

Logika opisowa w podstawowej swojej wersji nie wykorzystuje alternatywy ani negacji dlatego często się ją rozbudowuje o te pojęcia:
— $\neg$: zaprzeczenie pojęcia,
— $\sqcup$: suma dwóch pojęć.

Za pomocą konstruktorów oraz pojęć i relacji elementarnych można definiować pojęcia złożone, które wyróżniają się pewnymi charakterystycznymi cechami.

— Każda atomowa klasa jest klasą,
— Jeżeli R to rola i A to klasa to $\forall$R.A jest klasą,
— Jeżeli R to rola i a to stała to $\exists$R.a jest klasą,
— Jeżeli A, B to klasy to A $\sqcap$ B to klasa,
— Jeżeli A, B to klasy to A $\sqcup$ B to klasa,
— Jeżeli A, B to klasy to A $\sqsupseteq$ B to wyrażenie,
— Jeżeli A, B to klasy to A $\doteq$ B to wyrażenie,

Oprócz stosowania opisanych wyżej konstruktorów można rozszerzyć możliwość ekspresji logiki deskrypcyjnej stosując np. kwantyfikatory liczbowe (np.: „co najmniej trzy”, „ mniej niż cztery”), definiując zależności między rolami(np. rola R jest podzbiorem roli L, role odwrotne), itp.
Ontologia wyrażona za pomocą logiki deskrypcyjnej przedstawia strukturę i hierarchię pojęć z jakiejś dziedziny, pozwala na przedstawienie pewnego fragmentu wiedzy zdroworozsądkowej dotyczącej rzeczywistości. Mając grupę klas elementarnych:
człowiek, lekarz, dentysta, mężczyzna, kobieta, sportowiec, mistrz, mecz
oraz ról elementarnych:
leczy, gra, zwycięża
Można przedstawić hierarchię pojęć, zawieranie się w sobie oraz ich powią-

zania między sobą, np. chcąc wyrazić, że dentysta jest rodzajem lekarza można napisać:
lekarz $\sqsupseteq$ dentysta.
lekarz jest człowiekiem:
człowiek $\sqsupseteq$ lekarz.
W taki sposób można zdefiniować relację „is-a" czyli „jest". Logika opisowa pozwala oczywiście na definiowanie bardziej złożonych relacji np. człowiek to kobieta lub mężczyzna:
człowiek $\doteq$ kobieta $\sqcup$ mężczyzna
Można również zdefiniować, że lekarz to człowiek, który leczy jakiegoś człowieka:
lekarz $\doteq$ człowiek $\sqcap \exists$leczy.człowiek
Mistrz zaś to sportowiec, który zwycięża we wszystkich meczach:
mistrz $\doteq$ sportowiec $\sqcap \forall$zwycięża.mecz
Od strony formalnej interpretacja $\mathrm{I}\langle\Delta^I, \bullet^I\rangle$ w logice opisowej składa się z:
— niepustego zbioru obiektów Δ^I, czyli dziedziny,
— funkcji interpretacji $\bullet^I$, która odwzorowuje,
  — nazwę pojęcia w podzbiór A^I dziedziny Δ^I,
  — nazwę roli w relację binarną R^I, która jest podzbiorem $\Delta^I \times \Delta^I$,
  — nazwę obiektu w obiekt s^I będący elementem Δ^I.
Nieformalnie jako interpretację można traktować przypasanie rzeczywistych pojęć, relacji i obiektów ich odpowiednikom w ontologii.
Faktem, bądź wystąpieniem nazywa się formułę A(a), gdzie A jest klasą natomiast a jest obiektem. Fakt można rozumieć jako: „a jest wystąpieniem klasy A". Faktem też może być formuła w postaci R(a, B), gdzie R jest relacją a, b są faktami. W związku z tym można stwierdzić, że Tbox jest skończonym zbiorem pojęć a Abox jest skończonym zbiorem faktów.
Mówi się, że pojęcie jest spełnialne w pewnej interpretacji jeśli w tej interpretacji może mieć wystąpienia. W potocznym rozumieniu pojęcie jest spełnialne jeśli nie zawiera sprzeczności.
Wykorzystując koncepty i role atomowe oraz konstruktory można budować ontologie z dowolnych dziedzin.
Tbox i Abox, czyli pojęcia, role i wystąpienia można traktować jako bazę wiedzy o opisywanym przez nie świecie. Aby możliwe było sensowne wykorzystanie takiej bazy wiedzy konieczne jest opracowanie jakiegoś mechanizmu wnioskowania na tak opisanej wiedzy. Generalnie dla wiedzy wyrażonej za pomocą logiki deskrypcyjnej można wyróżnić kilka sposobów wnioskowania:
— Subsumcja – subsumpcja polega na odnajdywaniu klas ogólniejszych od zadanej. Dla tak zdefiniowanych dwóch klas:
  $\mathrm{B} \sqsubseteq \mathrm{A}$
  klasa A jest ogólniejsza od klasy B.

— Klasyfikacja – klasyfikacja polega na uporządkowaniu wszystkich klas ze względu na ogólność. Dla tak zdefiniowanych klas:
$B \sqsubseteq A$
tworzona jest hierarchia, w której klasa A jest nad klasą B (reprezentuje pojęcie ogólniejsze od pojęcia B). Klasyfikacja może również polegać na przypisaniu obiektów do poszczególnych klas na podstawie relacji w które wchodzą.
— Spełnialność – spełnialność polega na sprawdzaniu, czy kryteria przynależności do klasy są logicznie możliwe do spełnienia i znajdywaniu konceptów sprzecznych, np. jeśli w bazie wiedzy jest zapis:
$A \sqsubseteq B \sqcap A \sqcap \neg B$
to tak zdefiniowany koncept jest sprzeczny i dla danej interpretacji nie może wystąpić.
— Zgodność – zgodność polega na znajdowaniu pojęć tożsamych.

Bardziej szczegółowy i precyzyjny opis formalny logiki opisowej można znaleźć m.in. tu [4] a także w internecie.

3.6.2. Cele i narzędzia realizacji ontologii

Można wyróżnić kilka podstawowych celów jakie przyświecają konstruktorom ontologii:
— Współdzielenie wiedzy na dany temat między ludźmi i programami,
— Umożliwienie ponownego wykorzystania wiedzy,
— Rozdzielenie wiedzy dziedzinowej od sterowania (podstawowy paradygmat systemów ekspertowych),
— Umożliwienie analizy wiedzy dziedzinowej,

Logika opisowa (bądź jej różne modyfikacje) jest formalnym narzędziem do reprezentacji ontologii, jednakże w praktyce stosuje się komputerowe języki implementujące formalizmy logiki opisowej. Jednym z najpopularniejszych jest OWL (Web Ontology Language), który wykorzystując składnie XML jest rozszerzeniem języka RDF. Język ten został skonstruowany w celu umożliwienia realizacji Semantic Web, czyli sieci semantycznej – standardu opisu treści umieszczanych w internecie tak, by możliwe było nadanie im semantyki i aby możliwe było właściwe ich przetwarzanie, odpowiednie dla niesionego przez nie znaczenia. Pomimo tego, że standard OWL nie był w swoich zamierzeniach nastawiony na realizację systemów sztucznej inteligencji, to jego możliwości oraz solidne oparcie w formalnej logice okazały się bardzo przydatne do reprezentacji części wiedzy zdroworozsądkowej i specjalistycznej, szczególnie odnoszącej się do wykorzystywanych pojęć ich hierarchii i relacji.

Język OWL istnieje w trzech odmianach różniących się stopniem skomplikowania i ekspresją [18]:
— OWL Lite - najprostsza wersja języka pozwalająca na implementację niezbyt skomplikowanych taksonomii z mocno ograniczoną możliwością reprezentacji rozbudowanych ról,
— OWL DL - bardziej skomplikowana, zawierająca w sobie OWL Lite, o większej ekspresji, pozwala na przedstawienie wszystkiego co może wyrazić rozszerzoną nieco logiką opisową (stąd skrót DL: Description Logic),
— OWL Full - najbardziej rozbudowana, zawierająca w sobie OWL DL, pozwala m.in. na traktowanie klasy jako zbioru obiektów i jednocześnie jako obiekt. OWL Full ma największą ekspresję, w rzeczywistości jest on OWL DL pozbawionym różnego rodzaju ograniczeń wynikłych z trzymania się formalności logiki opisowej. Duża ekspresja języka skutkuje tym, że stał się on nierozstrzygalny.

Najczęściej stosowanym językiem jest OWL DL ze względu na maksymalną ekspresję przy zachowaniu rozstrzygalności.
Role w OWL DL mogą odnosić się nie tylko do innych klas ale także do konkretnych danych. Język definiuje następujące typy danych:
— Boolean,
— Number (int, float double itp),
— String,
— Enum,
— Instance,

Język pozwala również na wprowadzenie restrykcji (faset) dotyczących:
— zakresu wartości jakie może przyjąć dana powiązana jakąś rolą z obiektem klasy ,
— liczebności powiązań daną rolą,
— wartości zabronionych jakie nie mogą być powiązane daną rolą z obiektem danej klasy,
— najogólniejszej odpowiedniej klasy jaka może być powiązana daną rolą z obiektem danej klasy,
— grupy klas, jakie mogą być powiązane daną rolą z obiektem danej klasy,

Mechanizmy te pozwalają nieco rozszerzyć ekspresję klasycznej logiki opisowej. Dzięki nim za pomocą języka OWL DL można również m.in. wyrazić np. przechodniość relacji (krewny krewnego jest również krewnym), antysymetryczność (Zenek jest rodzicem Maćka w związku z czym Maciek jest dzieckiem Zenka), symetrię (Zenek jest krewnym Józka, w związku z czym Józek jest krewnym Zenka) itp. właściwości ról.
Poniżej zostanie przedstawiony przykład definicji jednego prostego pojęcia z ontologii dotyczącej wiedzy prawniczej w zakresie prawa o podatku rolnym:

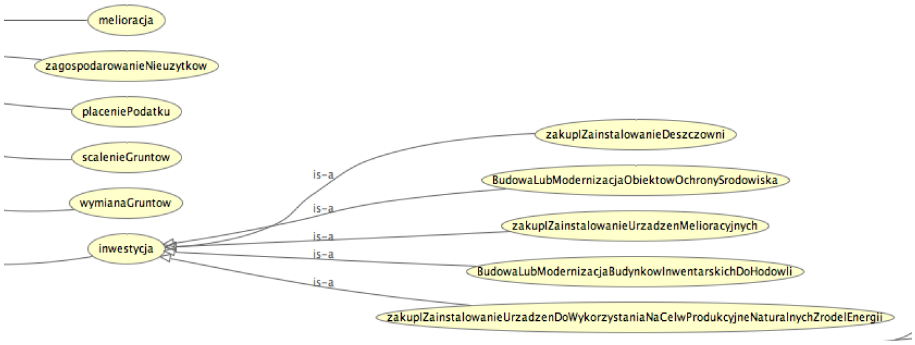
```
10  <SubClassOf>
        <Class URI="&swrl;scalenieGruntow"/>
12      <Class URI="&podatnik;Akcja"/>
    </SubClassOf>
14  <DisjointClasses>
        <Class URI="&swrl;scalenieGruntow"/>
16      <Class URI="&swrl;wymianaGruntow"/>
    </DisjointClasses>
```

Składnia powyższego zapisu niewątpliwie nie jest czytelnikowi obca gdyż język OWL opiera się na składni XML. Powyższy fragment definiuje klasę o nazwie *scalenieGruntow*, która jest podklasą klasy *Akcja*, jednocześnie definiuje, że jest to klasa rozłączna względem klasy *wymianaGruntów*, co oznacza, że żadne wystąpienie klasy *wymianaGruntow* nie może być równocześnie wystąpieniem klasy *scalenieGruntów*.
Przykładowa definicja roli w języku OWL wygląda tak:

```
10  <SubObjectPropertyOf>
        <ObjectProperty URI="&podatnik;jestOsobaFizyczna"/>
12      <ObjectProperty URI="&podatnik;jest"/>
    </SubObjectPropertyOf>
14  <ObjectPropertyDomain>
        <ObjectProperty URI="&podatnik;jestOsobaFizyczna"/>
16      <Class URI="&podatnik;Czlowiek"/>
    </ObjectPropertyDomain>
18  <ObjectPropertyRange>
        <ObjectProperty URI="&podatnik;jestOsobaFizyczna"/>
20      <Class URI="&podatnik;OsobaFizyczna"/>
```

Powyższy przykład definicji roli odnosi się do roli: *jestOsobaFizyczna*, która jest podzbiorem roli: *jest*. Rola *jestOsobaFizyczna* może występować między obiektem klasy *Czlowiek* a obiektem klasy *OsobaFizyczna*. Intuicyjnie można ją rozumieć jako powiązanie pewnej konkretnej osoby z jej pozycją wobec prawa.
Jednym z najpopularniejszych narzędziem do tworzenia ontologii za pomocą języka OWL DL jest program Protege (http://protege.stanford.edu/) oferowany na licencji Open Source i opracowany na Stanford University w Kalifornii. Pozwala on na prostą graficzną budowę ontologii: tworzenie klas, i ról atomowych i złożonych z wykorzystaniem szeregu konstruktorów i wszystkich wspomnianych wyżej mechanizmów.
Zanim powstanie ontologia należy odpowiedzieć sobie na kilka podstawowych pytań:
— Jaka jest dziedzina i co powinna zawierać?
— W jakim celu tworzy się ontologię?
— Jakie pytania do względem dziedziny mogą się pojawić?

— Kto będzie używał i utrzymywał ontologię?
Proces tworzenia ontologi składa się z:
— Definicji klas,
— Definicji hierarchii klas,
— Tworzenie złożonych klas,
— Definicja relacji,
— Definicji atrybutów i możliwych list wartości tych atrybutów,
— Tworzeniu instancji,
Ontologie wyrażone za pomocą OWL i logiki deskrypcyjnej (choć oczywiście nie tylko: istnieją również inne mechanizmy ich wyrażania, nawet wspomniane wcześniej ramy i sieci semantyczne mogą do tego posłużyć). można wykorzystywać do wielu różnych zastosowań: wszędzie tam, gdzie konieczne jest opisanie pewnej struktury i hierarchii pojęć z danej dziedziny. Oprócz, wspomnianej wcześniej, idei internetu semantycznego (na marginesie warto zauważyć, że jest to bardzo ostatnio popularny temat, który również, choć pośrednio, wiąże się ze sztuczną inteligencją) ontologie znajdują stosunkowo szerokie zastosowanie np. w procesie modelowania i opisu dokumentów prawnych i biznesowych, tam, gdzie wprowadza się ich elektroniczny obieg ale także tam gdzie konieczna jest możliwość szybkiego ich wyszukiwania, często przy nie do końca precyzyjnych jego kryteriach. Ontologia wiążąc je ze sobą w pewne tematyczne zakresy może ułatwić taki proces wyszukiwania odpowiednich dokumentów, aktów prawnych bądź poszczególnych przepisów. Stosuje się je również jako nośnik zdroworozsądkowej wiedzy przy różnego rodzaju chatbotach. Poniżej został zaprezentowany przykład fragmentu ontologii wspomagającej reprezentację wiedzy prawniczej w zakresie ustawy o podatku rolnym. Powyższy przykład pokazuje tylko zależności



Rysunek 3.4. Fragment przykładowej ontologii.

wynikające z relacji dziedziczenia (a dokładnie z relacji: „is-a” czyli „jest”). Na rysunku nie ma odwzorowanych innych relacji wynikających ze zdefinio-

wanych ról.

Ontologie pozwalają wyrazić wiedzę o, można to nazwać, statycznym charakterze, tzn. pozwalają opisać pewne istniejące zależności między klasami i bytami nie potrafią jednak odwzorować akcji, czyli zmian stanu faktycznego, zależności, reguł itp. W związku z czym z punktu widzenia realizacji systemów sztucznej inteligencji przydatne byłoby połączenie ontologii z innymi sposobami reprezentacji wiedzy. Niestety połączenie logiki opisowej z programowaniem logicznym mimo, że oba te systemy mieszczą się w logice pierwszego rzędu może stwarzać problemy obliczeniowe, dlatego jest to zadanie nieco bardziej skomplikowane i zazwyczaj wymaga ograniczenia ekspresji obu tych sposobów reprezentacji wiedzy (zainteresowanym autor proponuje poszukać w internecie informacji na temat tzw. reguł bezpiecznych oraz systemu reguł dedykowanego do OWL czyli SWRL).

Rozdział 4

Systemy uczące się

Jedną z najistotniejszych cech umysłowości ludzkiej jest zdolność do uczenia się. Oczywiście taką zdolność przejawiają nie tylko ludzie: każdy kto ma psa wie, że można go wielu rzeczy nauczyć, a czasem pewnych zachowań jest on nawet w stanie nauczyć się sam. Podobnie inne zwierzęta w mniejszym lub większym stopniu zdolność tą posiadają. Nie da się jednak zaprzeczyć, że zdolności umysłu ludzkiego w tej dziedzinie (i nie tylko tej) są bardzo wielkie.

Jest wiele sposobów uczenia się. Różne rodzaje wiedzy można zdobywać na różne sposoby: Czytając podręcznik student przyswaja sobie i zapamiętuje, lepiej, lub gorzej, zawartą w nim wiedzę. W przypadku rozwiązywania zadania student zdobywa wiedzę poprzez doświadczenie, próby i błędy jakie popełnia. Mechanik samochodowy zdobywa wiedzę na temat określonej marki samochodu wielokrotnie naprawiając różnego rodzaju usterki, dzięki czemu wie co najczęściej się psuje, jakie tego są objawy i jak można to naprawić. Ze względu na tematykę niniejszej pozycji interesujące dla czytelnika będą przede wszystkim takie sposoby uczenia się, które można zaimplementować w systemie komputerowym. Od biedy sam proces programowania można uznać za specyficzną formę uczenia komputera właściwych zachowań, ale oczywiście nie takie uczenie autor ma na myśli.

Warto w związku z tym zastanowić się czym tak naprawdę jest uczenie się w kontekście systemów informatycznych, na czym ono polega i jak można je zdefiniować. Autor najpopularniejszej (i najprawdopodobniej najlepszej) polskiej książki poświęconej systemom uczącym się [6] definiuje proces uczenia się:

> Uczeniem się systemu jest każda autonomiczna zmiana zachodząca w systemie na podstawie doświadczeń, która prowadzi do poprawy jakości jego działania

Przyglądając się powyższej definicji łatwo zauważyć, że autor zawęził ją do procesu uczenia się przez doświadczenie, oraz wykluczył przypadki, w których wiedza jest wprowadzana bezpośrednio do systemu (jak jest np. w systemach ekspertowych). Pomimo tego pole wykorzystania tego typu narzędzi jest ogromne: przede wszystkim tego rodzaju systemy mogą pomagać w odkrywaniu nieznanych zasad i reguł, przez co mają duże pole do wykorzystania w procesie eksploracji danych. Poza tym systemy uczące mogą być przydatne w sytuacjach, w których wymagana jest samodzielność systemu i umiejętność dostosowywania się do istniejących a nieznanych wcześniej warunków. Realizacja systemów uczących się nie jest oczywiście żadną alternatywą dla np. tradycyjnego procesu programowania: systemy uczące stosuje się nie tam, gdzie są trudności z procesem programowania a

tam, gdzie jest brak wiedzy jak dane zadanie można rozwiązać.

Warto przyjrzeć się w tym punkcie do jakich celów można wykorzystać, lub już wykorzystuje się systemy uczące się:
— generacja wiedzy deklaratywnej w postaci drzew decyzyjnych i reguł,
— definiowanie nowych pojęć,
— wykrywanie trendów, tendencji i innych prawidłowości w danych.
Teoria i praktyka realizacji systemów uczących się wypracowała szereg algorytmów pozwalających na realizację systemów uczących się. Nie każdy z nich nadaje się oczywiście do każdego zadania. Dość obszerny opis bardzo wielu z nich czytelnik może znaleźć w [6].Wiele z nich wykorzystuje się w procesach eksploracji danych, które poruszone będą nieco później Poniżej zostanie zaprezentowany, chyba, najpopularniejszy z algorytmów wykorzystywanych w maszynowym uczeniu się: Indukcyjny algorytm generacji drzew decyzyjnych.

4.1. Algorytm ID3

Jednym z najpopularniejszych algorytmów pozwalających na realizację systemów uczących się jest algorytm ID3 opracowany przez Quinlana [20]. Algorytm ten służy do indukcyjnego pozyskiwania wiedzy deklaratywnej (w postaci drzewa decyzyjnego).
Generalnie wnioskowanie indukcyjne opiera się na przechodzeniu od obserwacji jednostkowych do ogólniejszych wniosków. W przypadku systemów komputerowych wnioskowanie indukcyjne opiera się na generacji pewnych ogólnych zasad na podstawie istniejących danych, obserwacji, przypadków itp.
Algorytm ID3 pozwala na generację drzewa decyzyjnego na podstawie szeregu przypadków jednostkowych. Drzewo decyzyjne jest w tym wypadku pewną strukturalnym zapisem wiedzy, pozwalającym na podstawie wartości pewnych cech (warunkowych) przypisać konkretne wartości cechom decyzyjnym.
Bardziej formalnie drzewo decyzyjne to struktura złożona z węzłów, z których wychodzą gałęzie prowadzące do innych węzłów lub liści, lub inaczej: drzewo decyzyjne to dowolny spójny skierowany graf acykliczny, gdzie krawędzie są nazywane gałęziami, wierzchołki, z których wychodzą gałęzie nazywane są węzłami a pozostałe wierzchołki nazywane są liśćmi [6]. Węzły w drzewie decyzyjnym wyrażają test na wartość jakiegoś atrybutu, gałęzie wychodzące z tego węzła wyrażają poszczególne wartości analizowanego atrybutu, liście zaś reprezentują kategorie decyzyjne. Czytelnicy, których interesuje pełna formalna definicja drzewa decyzyjnego mogą ją znaleźć np.

tu [6].

Aby wygenerować za pomocą algorytmu ID3 drzewo decyzyjne konieczny jest stosunkowo liczny zbiór przykładów opisujących daną sytuację. Każdy przykład ze zbioru przyjmuje jakąś wartość dla każdego atrybutu z listy atrybutów warunkowych oraz atrybutu decyzyjnego. Każdy atrybut opisujący dany przykład może przyjąć jedną z listy (osobnej dla każdego atrybutu) możliwych wartości. Taki zbiór przykładów nazywa się zbiorem uczącym.
W większości pozycji literaturowych zasada działania algorytmu ID3 objaśniana jest na przykładzie wpływu warunków atmosferycznych na to, czy ktoś decydował się czy wyjść grać w golfa, czy też nie [20],[6],[13]. Poniżej zostanie zaprezentowany zbiór uczący do tego przykładu: Problematyka

Tabela 4.1. Zbiór uczący

Lp.	Pogoda	Temperatura	Wilgotność	Wiatr	Golf
1	słońce	wysoka	wysoka	nie	nie
2	słońce	wysoka	wysoka	tak	nie
3	zachmurzenie	wysoka	wysoka	nie	tak
4	deszcz	średnia	wysoka	nie	tak
5	deszcz	niska	normalna	nie	tak
6	deszcz	niska	normalna	tak	nie
7	zachmurzenie	niska	normalna	tak	tak
8	słońce	średnia	wysoka	nie	nie
9	słońce	niska	normalna	nie	tak
10	deszcz	średnia	normalna	nie	tak
11	słońce	średnia	normalna	tak	tak
12	zachmurzenie	średnia	wysoka	tak	tak
13	zachmurzenie	wysoka	normalna	nie	tak
14	deszcz	średnia	wysoka	tak	nie

grania w golfa (bądź nie) przy określonych warunkach pogodowych opisana jest przez 5 atrybutów, z których każdy może przyjmować wartości ze skończonego i danego zbioru:
— Atrybut: Pogoda, zbiór wartości: *słońce*, *deszcz*, *zachmurzenie*.
— Atrybut: Temperatura, zbiór wartości: *wysoka*, *średnia*, *niska*.
— Atrybut: Wilgotność, zbiór wartości: *wysoka*, *normalna*.
— Atrybut: Wiatr, zbiór wartości: *tak*, *nie*.
— Atrybut: Golf, zbiór wartości: *tak*, *nie*.
Ponieważ celem jest zbadanie, przy jakich warunkach atmosferycznych chodzi się grać w golfa, to atrybutem decyzyjnym jest Golf, który ma dwa możliwe stany: *tak* i *nie*.
Gdy dana jest lista atrybutów wraz z listami dostępnych wartości oraz zbiór

uczący, można rozpocząć budowę drzewa decyzyjnego.
Zasada działania algorytmu ID3 opiera się na tym, że wybiera się wg jakiegoś (za chwile zostanie sprecyzowane jakiego) klucza atrybut, następnie traktuje się go jako pierwszy węzeł, z którego wychodzić będzie tyle gałęzi ile wartości może przyjąć ten atrybut. Każda z gałęzi wyraża wybór którejś z wartości analizowanego atrybutu. Na końcu każdej gałęzi tworzy się nową listę przykładów, taką dla której atrybut nadrzędny ma taką wartość jaką wyraża prowadząca z niego gałąź. W tym punkcie znów dobiera się wg wspomnianego wyżej klucza atrybut, który będzie testowany w kolejnym węźle, chyba, że zaistnieje warunek stopu, wtedy na końcu gałęzi wstawia się liść, który oznacza kwalifikację wszystkich przykładów spełniających powyższe warunki do jednej kategorii atrybutu decyzyjnego. Gdy na końcu gałęzi jest nowy węzeł wyprowadza się z niego tyle gałęzi ile możliwych wartości atrybut powiązany z tym węzłem może przyjąć itd.
Na końcu pozostaje zasadnicze kryterium doboru atrybutów: który z nich będzie rozpatrywany jako korzeń, które zaś będą wstawiane dalej. W tym punkcie Quinlan zaproponował wykorzystanie kryterium względnego maksymalnego przyrostu informacji. Do wyznaczenia owego względnego maksymalnego przyrostu informacji konieczne jest wyznaczenie ilości informacji, czyli entropii oraz entropii zbioru przykładów ze względu na analizowany atrybut.
Entropia:

$$I(E) = -\sum_{i=1}^{k} \frac{\mid E_i \mid}{\mid E \mid} \cdot \log_2 \left(\frac{\mid E_i \mid}{\mid E \mid} \right) \tag{4.1}$$

gdzie:
k – Liczba wartości atrybutu decyzyjnego
$\mid E_i \mid$ – Liczba przykładów ze zbioru uczącego mających i-tą wartość atrybutu decyzyjnego
Entropia zbioru przykładów ze względu na analizowany atrybut:

$$I(E,a) = -\sum_{m=1}^{m=L} \frac{\mid E^{(m)} \mid}{\mid E \mid} \cdot I\left(E^{(m)}\right) \tag{4.2}$$

a – analizowany atrybut
L – liczba wartości analizowanego atrybutu
$E^{(m)}$ – przykłady, dla których a-ty atrybut miał m-tą wartość
$\mid E^{(m)} \mid$ – liczba przykładów, dla których a-ty atrybut miał m-tą wartość
Względny maksymalny przyrost informacji:

$$\triangle I(E,a) = I(E) - I(E,a) \tag{4.3}$$

Atrybut, dla którego względny maksymalny przyrost informacji będzie największy będzie wybrany jako pierwszy węzeł (korzeń) drzewa decyzyjnego.

W pierwszym kroku należy zatem wyznaczyć entropię dla całego zbioru przykładów. Ponieważ wśród 14 przykładów w 9 z nich atrybut decyzyjny był na tak (chodzono grać w golfa), a w 5 na nie, to z wzoru 4.1 entropia wynosi:

$$I(E) = -\frac{9}{14}\log_2\frac{9}{14} - \frac{5}{14}\log_2\frac{5}{14} = 0,940$$

Następnie należy oczekiwaną ilość informacji w zbiorze przykładów ze względu na każdy atrybut. Pierwszym analizowanym atrybutem będzie pogoda. Zgodnie ze wzorem 4.2 do wyznaczenia tej wielkości konieczne jest wyznaczenie entropii dla podzbiorów przykładów dla wszystkich wartości atrybutu pogoda. W zbiorze przykładów jest łącznie 5 przykładów, w których wartość trybutu pogoda była równa „słońce", w związku z czym:

$$I(E^{slonce} = -\frac{2}{5}\log_2\frac{2}{5} - \frac{3}{5}\log_2\frac{3}{5} = 0,971$$

analogicznie dla reszty wartości atrybutu pogoda, entropia wynosi:

$$I(E^{zachmurzenie} = -\frac{4}{4}\log_2\frac{4}{4} - \frac{0}{4}\log_2\frac{4}{4} = 0$$

$$I(E^{deszcz} = -\frac{3}{5}\log_2\frac{3}{5} - \frac{2}{5}\log_2\frac{2}{5} = 0,971$$

W związku z tym, oczekiwana łączna ilość informacji ze względu na atrybut pogoda będzie wynosić:

$$I(E, pogosa) = \frac{5}{14}I(E^{slonce}) + \frac{4}{14}I(E^{zachmurzenie}) + \frac{5}{14}I(E^{deszcz}) =$$

$$\frac{5}{14}\cdot 0,971 + \frac{4}{14}\cdot 0 + \frac{5}{14}\cdot 0,971 = 0,694$$

Wreszcie względny maksymalny przyrost informacji:

$$\triangle I(E, pogoda) = I(E) - I(E, pogoda) = 0,940 - 0,694 = 0,246$$

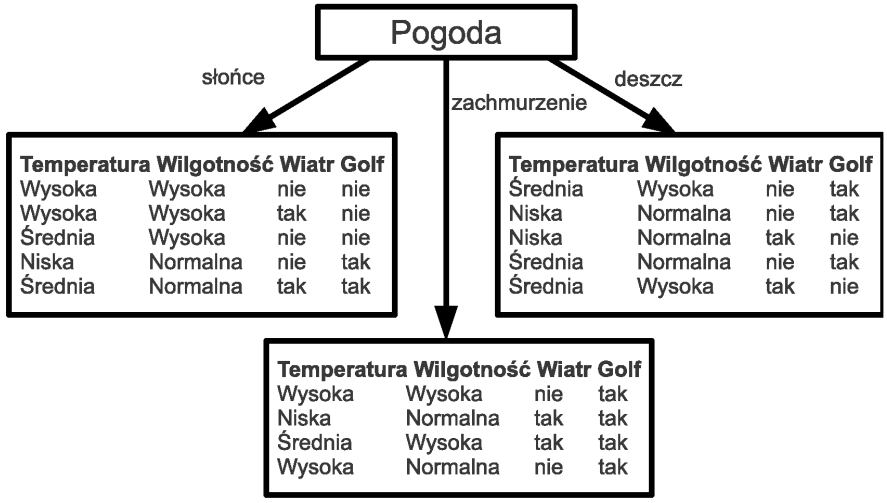
Analogicznie dla innych atrybutów:

$$\triangle I(E, temperatura) = 0,029$$

$$\triangle I(E, wilgotnosc) = 0,151$$

$$\triangle I(E, wiatr) = 0,048$$

Z powyższych obliczeń wynika, że pierwszym węzłem w drzewie decyzyjnym będzie węzeł reprezentujący atrybut pogoda. W związku z tym może powstać pierwszy węzeł drzewa decyzyjnego: Z węzła wychodzą trzy gałęzie oznaczające wybór jednej z trzech możliwych wartości atrybutu pogoda:



Rysunek 4.1. Pierwszy węzeł drzewa decyzyjnego.

słońce, *zachmurzenie*, *deszcz*. Na końcu każdej gałęzi utworzone są nowe tabelki, które zawierają wszystkie przykłady, dla których wartość atrybutu pogoda jest taka jak wybrana gałąź.
W kolejnym kroku sprawdza się, czy nie ma gałęzi, w której wszystkie przykłady mają tą samą wartość atrybutu decyzyjnego. Jeśli tak jest na końcu gałęzi umieszcza się liść, który oznacza kwalifikację do określonej kategorii atrybutu decyzyjnego. W analizowanym przypadku, gdy atrybut Pogoda przyjmie wartość *zachmurzenie*, wtedy wartość atrybutu Golf wynosi zawsze *tak*, co można interpretować tak, że gdy pogoda jest zachmurzona to zawsze chodzi się grać w golfa.
W kolejnym kroku, dla pozostałych gałęzi, analogicznie jak było dla pierwszego węzła wyznacza się atrybuty (osobny dla każdej gałęzi), które będą kolejnymi węzłami.
W związku z tym dla pierwszej gałęzi (*słońce*):

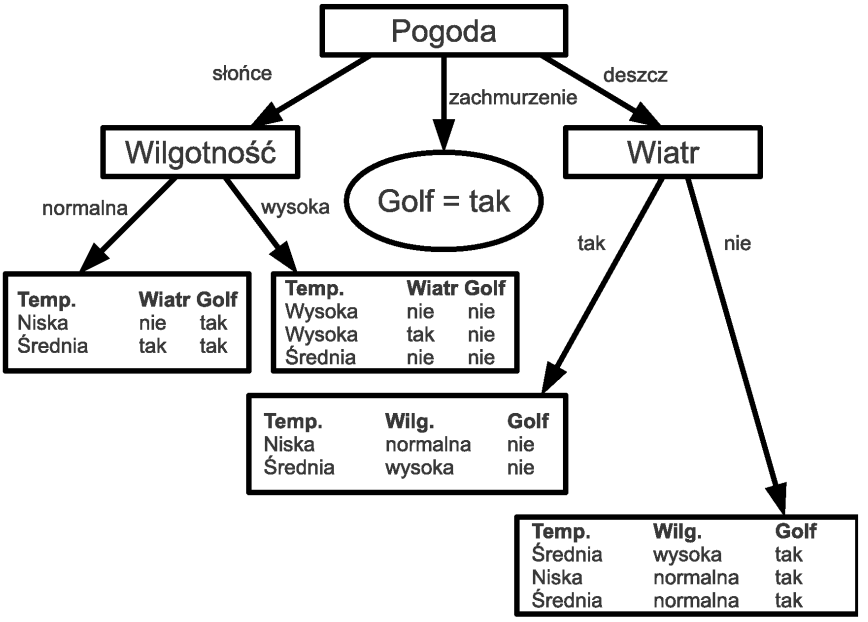
$$\triangle I(E, temperatura) = 0,571$$

$$\triangle I(E, wilgotnosc) = 0,971$$

$$\triangle I(E, wiatr) = 0,02$$

Największy względny maksymalny przyrost informacji jest dla atrybutu Wilgotność i węzeł na końcu gałęzi *słońce* będzie wyrażał test na wilgotność. Wychodzić z niego będą dwie gałęzie: *wysoka* i *normalna*.

Dla trzeciej gałęzi (*deszcz*) atrybutem o największym względnym maksymalnym przyroście informacji jest atrybut Wiatr, w związku z czym będzie on reprezentował test na wartość tego atrybutu: wychodzić z niego będą dwie gałęzie: *tak* i *nie*. W tym punkcie drzewo decyzyjne będzie wyglądać tak jak na rysunku 4.2 Łatwo zauważyć, że wszystkie tabelki z przykłada-

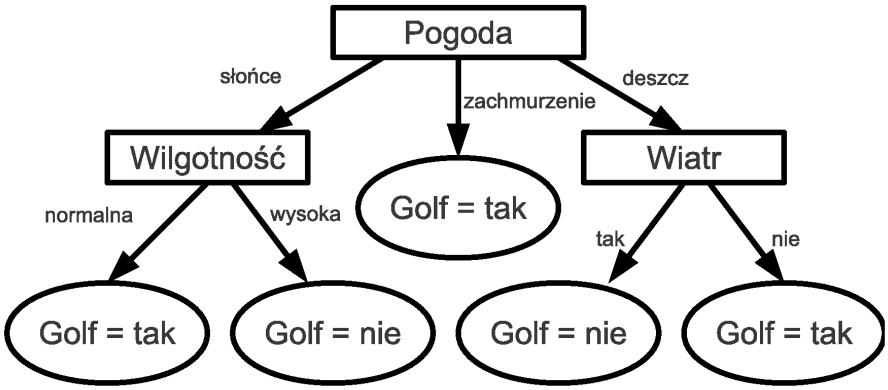


Rysunek 4.2. Drugi poziom węzłów drzewa decyzyjnego.

mi na końcu gałęzi jednoznacznie kwalifikują je do odpowiedniej wartości atrybutu decyzyjnego. W związku z tym można zakończyć budowę drzewa decyzyjnego, którego ostateczna postać jest na rysunku 4.3 Z takiego drzewa decyzyjnego bardzo łatwo jest przejść do regułowej formy reprezentacji wiedzy:

Golf = *tak* **Jeżeli** Pogoda = *słońce* **i** Wilgotność = *normalna*
Golf = *nie* **Jeżeli** Pogoda = *słońce* **i** Wilgotność = *wysoka*
Golf = *tak* **Jeżeli** Pogoda = *zachmurzenie*
Golf = *nie* **Jeżeli** Pogoda = *deszcz* **i** Wiatr = *tak*
Golf = *tak* **Jeżeli** Pogoda = *deszcz* **i** Wiatr = *nie*

Zasada działania algorytmu ID3 jest bardzo prosta: kryterium maksymalnego przyrostu informacji pozwala na wybranie kolejności rozpatrywania



Rysunek 4.3. Drzewo decyzyjne.

poszczególnych atrybutów w drzewie. Owa kolejność potrzebna jest przede wszystkim do tego by maksymalnie uprościć drzewo decyzyjne. Gdyby korzeniem drzewa zrobić np. atrybut Wilgotność, wtedy byłoby ono znacznie bardziej skomplikowane i trudne do interpretacji.

4.2. Wady, zalety i rozwinięcie algorytmu ID3

Warto teraz przyjrzeć się przedstawionemu w poprzednim rozdziale algorytmowi jakie są jego zalety?
Bardzo ważną zaletą algorytmu ID3 jest to, że jeśli dane są poprawne i kompletne to zawsze da poprawne wyniki. Poza tym jest to algorytm stosunkowo szybki, nie wymagający czasochłonnych obliczeń.
Niestety algorytm ID3 nie jest bez wad. Część z nich widoczne jest już na pierwszy rzut oka: przede wszystkim algorytm ten nie radzi sobie gdy zbiór przykładów jest niekompletny, tzn. gdy w przykładach są luki, bądź dane są zaszumione, czyli dla takich samych wartości atrybtutów warunkowych są różne wartości atrybutu decyzyjnego. Algorytm ID3 działa tylko na wartościach dyskretnych, czyli nie potrafi wygenerować drzewa decyzyjnego dla wartości ciągłych. Inną dość charakterystyczną wadą jest możliwość budowy, przy dużej liczbie przykładów, zbyt dużych drzew decyzyjnych, które z jednej strony będą trudne do interpretacji a z drugiej, przy nieco zaszumionych danych mogą dawać nie zawsze poprawne wyniki.
Większość wspomnianych wyżej wad można jednak naprawić modyfikując nieco algorytm:
— Problem niekompletnych danych można rozwiązać na kilka sposobów:

  - Jeśli luk nie ma wiele to można przykłady z brakującymi danymi po prostu usunąć ze zbioru uczącego. Jest to rozwiązanie najprostsze ale też najmniej skuteczne.
  - Można brakujące dane wylosować, bądź w bardziej rozbudowanej wersji wylosować bazując na prawdopodobieństwach z jakimi pojawiają się poszczególne wartości danego atrybutu.
  - Można wpisać najczęściej pojawiającą się wartość danego atrybutu,
  - oraz wiele innych...
- Wartości ciągłe można skwantyfikować. Ważne w tym punkcie jest to, by kwantyfikacja oddawała właściwe proporcje danych. Dla przykładu, gdyby jednym z atrybutów była temperatura ciała człowieka i skwantyfikowano ją na trzy wartości: *wysoka*, *średnia* i *niska* wg temperatur: do 15 stopni celsjusza niska, między 20 a 30 stopni średnia i powyżej 30 wysoka, to praktycznie zawsze człowiek mieści się w zakresie temperatury wysokiej i takiego atrybut nie można by użyć do generacji drzewa.
- Na zaszumienie danych najlepszym rozwiązaniem jest przycięcie drzewa. Sam problem zaszumienia danych może wynikać np. z pewnej losowości w zbiorze przykładów (np. raz się zdarzyło, że ktoś poszedł grać w golfa przy niesprzyjającej temu pogodzie). Przycinanie drzewa polega na skróceniu zbyt długich gałęzi i wstawieniu na końcu wybranej gałęzi liścia zamiast węzła. W takim liściu przypisuje się taką wartość atrybutu decyzyjnego jaka najczęściej pojawia się w opisującym go zbiorze przykładów. Ważny jest oczywiście wybór punktu przycięcia drzewa, jedną z metod wybory takiego punktu może być np. odpowiedni udział procentowy jednej wartości, np.: 90-10.
- Przycinanie drzewa jest oczywiście dobrą metodą ograniczania zbyt dużego drzewa, dla którego zbyt dokładne dopasowanie nie ma sensu (np. ze względu na losowe zaszumienie).

Sam autor algorytmu ID3, Ross Quinlan po kilku latach przedstawił jego modernizację w podobny sposób rozwiązującą wspomniane wyżej problemy, czyli algorytm C4.5 [21].

Algorytm ten w sytuacji braku pewnych danych pomijał je przy liczeniu przyrostu informacji.

Dla danych ciągłych algorytm wybiera taki punkt podziału na wartości dyskretne aby przyrost informacji był w nim największy.

W algorytmie C4.5 wykorzystano również specjalny mechanizm przycinania drzewa bazujący na porównywaniu przewidywanego błędu danego węzła z błędem klasyfikacji, gdyby zastąpić ten węzeł liściem.

Rozdział 5

Programowanie w logice z ograniczeniami

Uważny czytelnik niewątpliwie zwrócił uwagę, że opisany wcześniej język PROLOG znakomicie nadawał się do modelowania problemów przeszukiwania. Notacja języka pozwala w prosty, spójny i czytelny sposób wyrazić domenę i ograniczenia przestrzeni przeszukiwania bez wnikania w techniczną stronę samego algorytmu. Niestety algorytm standard backtracking zaimplementowany w kompilatorach języka PROLOG, mimo iż skuteczny w rozwiązywaniu problemów logicznych, nie jest wystarczający dla problematyki przeszukiwania, w której zazwyczaj do „przejrzenia" jest bardzo duża liczba stanów. W związku z tym w połowie lat osiemdziesiątych wpadnięto na pomysł aby rozszerzyć możliwości klasycznego PROLOG-a i stworzyć nową klasę języków bazujących na jego paradygmacie i notacji, ale lepiej znajdujących się w problematyce przeszukiwania i optymalizacji.

Bazując na powyższych założeniach powstało kilka języków programowania, które w różny (lepszy lub gorszy) sposób wykorzystują PROLOG jako bazę narzędzia pozwalającego rozwiązywać problemy kombinatoryczne. Języki te nazwano językami *Constraint Logic Programming*, czyli programowania w logice z ograniczeniami. Dlaczego z ograniczeniami? ano dlatego, że istotą tego rodzaju problemów jest taki proces przeszukiwania przestrzeni aby wypełnić szereg postawionych przez użytkownika ograniczeń zawężających domenę przeszukiwania i utrudniających znalezienie odpowiedniego rozwiązania. Dodatkowo w językach CLP zaimplementowane są mechanizmy optymalizacji, czyli poszukiwania najlepszych (najmniejszych lub największych wartości) wg zdefiniowanej przez programistę funkcji celu. Podsumowując: języki CLP łączą w sobie tradycyjny paradygmat programowania w logice, rozbudowane mechanizmy spełniania ograniczeń oraz wydajne mechanizmy optymalizacyjne, przez co są znakomitymi narzędziami do rozwiązywania problemów przeszukiwania kombinatorycznego i optymalizacji, tradycyjnie już utożsamianych ze sztuczną inteligencją.
Niniejsza praca, nie aspiruje do bycia kompletnym kursem przykładowego języka CLP. Celem tego rozdziału jest przedstawienie podstawowych założeń oraz notacji języka wraz z podstawowymi algorytmami zaimplementowanymi w kompilatorze. Osobom bardziej zainteresowanym można zalecić jedyną w Polsce, ale za to bardzo dobrze napisaną pozycję [17]. Najpopularniejszym językiem programowania klasy CLP jest język ECLiPSe, rozpowszechnianym na licencji *Mozilla-Style Public License* i dostępnym pod adresem: www.eclipseclp.org. Jego popularność wynika z jego dużej wydajności oraz, oczywiście, braku opłat licencyjnych.
Wszelkie szczegóły związane z instalacją i obsługą kompilatora bez trudu można znaleźć w dokumentacji (www.eclipseclp.org), dlatego rozdział ten skoncentruje się na tym jakie problemy i w jaki sposób można rozwiązać za

pomocą języka ECLiPSe.

5.1. Informacje podstawowe

Na wstępie warto przyjrzeć się ogólnym zasadom pisania programów w języku ECLiPSe. Przede wszystkim dobrze jest mieć świadomość do rozwiązywania jakiej klasy problemów język ten został stworzony. Uważny czytelnik już zauważył, że głównymi problemami do których się go wykorzystuje są zadania przeszukiwania i optymalizacji. Paradygmat programowania w logice z ograniczeniami zakłada połączenie ze sobą trzech elementów: programowania w logice, przeszukiwania z rozwiązywaniem ograniczeń i optymalizacji. Wspomniane wcześniej ograniczenia, to relacje, które muszą być spełnione i które zawężają przestrzeń przeszukiwania. Na czym polega owo rozwiązywanie ograniczeń? Na wstępie deklaruje się zbiór zmiennych, następnie zbiór dopuszczalnych wartości, czyli domeny zmiennych, na końcu zaś zbiór ograniczeń. Jakiego rodzaju to mogą być relacje? Najprostsze z nich polegają na sztywnym ograniczeniu domeny, np. wartość zmiennej X<5 - oznacza to, że wartość zmiennej X musi być zawsze mniejsza od 5. Większość rzeczywistych problemów wymaga jednak bardziej wyrafinowanych ograniczeń tzn. takich, w których występuje więcej, powiązanych ze sobą zmiennych, np. X > Y+5. W takiej sytuacji dopuszczalny zakres wartości zmiennej X jest zależny od wartości przypisanej do zmiennej Y i gdy Y=3, to wartość X musi być większa od 8 a gdy Y=4 to wartość X musi być większa od 9. Język ECLiPSe pozwala na stosowanie domeny ciągłej, dyskretnej, stosowania list dopuszczalnych wartości itp. W klasycznym PROLOG-u wykorzystany jest algorytm standard backtracking w językach CLP wykorzystano algorytm Forward checking bądź Forward checking + looking ahead, które są znacznie wydajniejsze od standard backtracking. Ich istota sprowadza się do propagacji ograniczenia i aktualizacji domeny po każdym ukonkretnieniu każdej zmiennej (dokładniej zostaną opisane nieco później). Do kompilatora języka ECLiPSe można dołączyć również szereg solwerów posługujących się algorytmami optymalizacyjnymi działającymi w domenach ciągłych lub dyskretnych.

Do jakich problemów wykorzystuje się języki CLP? Przede wszystkim do całej klasy zadań związanych z różnego rodzaju harmonogramowaniem: układanie planów zajęć, prac maszyn, zadań dla pracowników, często jest to połączone z optymalizacją (minimalizacja kosztów pracy maszyn, przerw i przestojów w pracy, luk między zajęciami itp.), podobnie języki CLP wykorzystuje się w rozwiązywaniu problemów transportowych (problem komi-

wojażera, minimalizacja kosztów przewozu towaru z i do różnych miejsc), problemu alokacji zasobów (np. inwestycje kapitałowe), problem pakowania elementów o nieregularnych kształtach w zamkniętych przestrzeniach (jak ułożyć przedmioty w kontenerze aby zmieściło ich się maksymalnie dużo, zadanie to można strywializować do problemu pakowania się z całą rodziną na wyjazd wakacyjny: jak zmieścić do „malucha" cztery osoby, trzy walizki, ponton, dwa leżaki, namiot i kuchenkę gazową), problem rozkroju (jak pociąć materiał o jakimś kształcie i powierzchni aby dało się z niego zrobić jak najwięcej wyrobów) oraz wiele wiele innych. Praktycznie każde zadanie związane z optymalizacją można rozwiązywać za pomocą języka CLP. Warto zwrócić jednak uwagę, że w narzędziu tym nie zrobi się nic więcej, niż np. w języku C++, jego siła jest w zwięzłości i czytelności programów, w których sam model problemu jest programem go rozwiązującym, dzięki czemu jest on bliższy intuicyjnemu rozwiązywaniu tego rodzaju zadań.

5.2. Pierwszy program

Jak w związku z tym wyglądają programy w języku ECLiPSe? Przede wszystkim zasady programowania są łudząco podobne do języka PROLOG, czyli jest to język deklaratywny: definiuje się fakty, reguły a po kompilacji można zadawać zapytania. Składnia języka również jest bardzo zbliżona do j. PROLOG i większość prologowych programów będzie działać także w języku ECLiPSe. To co jest charakterystyczne to bardzo rozbudowany zestaw predykatów standardowych służących do definiowania domeny, ograniczeń a także uruchamiających procesy przeszukiwania i optymalizacji. Generalnie program w języku ECLiPSe składa się z:

— deklaracji bibliotek,
— deklaracji faktów,
— deklaracji reguł

  Reguły w programach są zbliżone w formie do reguł w PROLOG-u, z tym że ponieważ programy w języku ECLiPSe zazwyczaj dotyczą problemów przeszukiwania i optymalizacji dlatego reguły zwykle mają charakterystyczną budowę i składają się z:
  — deklaracji zmiennych,
  — deklaracji domen poszczególnych zmiennych,
  — definicji ograniczeń,
  — definicji parametrów sterujących (opcje przeszukiwania ewentualnie optymalizacji itp.

Strukturę programu najlepiej będzie można zrozumieć przyglądając się jakiemuś przykładowemu programowi.

Poniżej zaprezentowano prosty program rozwiązujący angielską zagadkę:

Należy pod każdą literę podstawić taką cyfrę, by prawdziwe było równanie:
SEND + MORE = MONEY

Listing 5.1. Program w języku ECLiPSe rozwiązujący problem SEND+MORE

```
:- lib(ic).
sendmore(Digits) :-
Digits = [S,E,N,D,M,O,R,Y],
Digits :: [0..9],
alldifferent(Digits),
S #\= 0,
M #\= 0,
1000*S + 100*E + 10*N + D
+ 1000*M + 100*O + 10*R + E
#= 10000*M + 1000*O + 100*N + 10*E + Y,
labeling(Digits).
```

Jak działa powyższy program? Pierwszy linijka to deklaracja biblioteki, biblioteka ic to podstawowa biblioteka służaca do przeszukiwania, działa tak w domenie liczb całkowitych i rzeczywistych. Główny predykat zdefiniowany w programie to *sendmory(Digits)* i jest on opisany przez koniunkcję warunków: W linii trzeciej przypisano zmiennej Digits listę zmiennych, każda z nich to jedna litera z zagadki. W linii czwartej zadeklarowano domenę zmiennych (każda zmienna w liście Digits przyjmuje wartość od 0 do 9), w piątej linii predykatem standardowym *alldifferent* narzucono, że wartość każdej zmiennej w liście powinna być inna. W kolejnych linijkach są ograniczenia o charakterze arytmetycznym: w szóstej i siódmej jest zastrzeżenie, że zmienne S i M powinny być różne od zera, natomiast linijki 8-10 opisują samą łamigłówkę. Warto tu zwrócić uwagę na znak #, który oznacza, że mechanizm wnioskowania powinien wartości w tych ograniczeniach uzgadniać dla domeny liczb całkowitych, gdyby zamiast znaku # był znak $ oznaczałoby to zmienne powinny przyjmować wartości rzeczywiste. Wreszcie w ostatniej, jedenastej linii jest wykorzystany standardowy predykat *labelling*, który informuje kompilator, które zmienne ma ukonkretniać.
Po kompilacji można zadać zapytanie: „sendmory(Digits).”, na które system poda prawidłową odpowiedź (jaką? proponuję sprawdzić).

5.3. Algorytmy

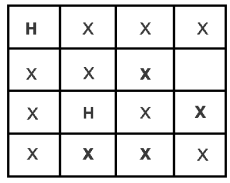
Skoro już powstał pierwszy program warto przyjrzeć się temu co jest w tle i jak działają algorytmy przeszukiwania w języku ECLiPSe. We jednym z wcześniejszych rozdziałów dość szczegółowo został omówiony algorytm

standard backtracking oraz jego wyższość nad przeglądem zupełnym. I tak jak standard backtracking jest w PROLOG-u tak algorytm forward checking jest w językach CLP. Algorytm ten jest rozwinięciem standard backtracking różni się tym, że przyczyną nawrotu nie jest naruszenie ograniczenia a brak dostępnych możliwości ruchu, gdyż po każdym ukonkretnieniu aktualizowana jest domena każdej zmiennej. Najłatwiej zasadę działania przedstawić na przykładzie z książki [17]. Problem jest prosty i powszechnie znany: Tak ustawić 4 hetmany na szachownicy 4X4 aby się nie biły. Na początku warto się przyjrzeć jak zakodować opisany tu przypadek. Najprostszym sposobem jest wykorzystanie czteroelementowej listy, w której pozycja w liście oznacza kolumnę, natomiast wartość oznacza wiersz w którym jest hetman. Ponieważ nie da się postawić dwóch hetmanów w jednej kolumnie to taki sposób zakodowania szachownicy jest wystarczający. Jak działa algorytm?
Na wstępie jest czteroelementowa lista z nieukonkretnionymi wartościami zmiennych: [K1,K2,K3,K4] każda zmienna ma domenę od 1 do 4 (liczby całkowite). W pierwszym kroku jest ukonkretniana pierwsza zmienna: [1,K2,K3,K4] oraz uaktualniane są domeny reszty zmiennych, czyli z ich domen usuwane są te wartości, które naruszają ograniczenia, w przypadku problemu hetmanów usuwane są te wartości, które reprezentują ustawienie w jednej linii lub na ukos względem już ustawionych hetmanów. Ilustruje to rys. 5.1 (H - hetman, X - pole zabronione).
W kolejnym kroku system podejmuje próbę ukonkretnienia kolejnej zmien-

H	X	X	X
X	X		
X		X	
X			X

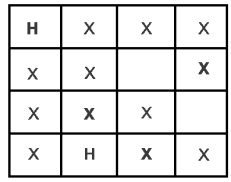
Rysunek 5.1. Ustawienie pierwszego hetmana.

nej, czyli K2 pierwszą wartością z domeny. Ponieważ domena została już uaktualniona to pierwsza wartość jaką może przyjąć zmienna to 3:[1,3,K3,K4]. Ukonkretnienie tej zmiennej powoduje ponowne uaktualnienie domen (rys. 5.2).
Niestety po aktualizacji domeny zmiennej K3 okazuje się, że jest ona pusta (nie ma możliwości wstawienia hetmana), w związku z czym system musi dokonać nawrotu do poprzedniego stanu i poszukać innej wartości zmiennej K2. Ważne tu jest to, że przyczyną nawrotu nie jest już naruszenie ograniczenia (wstawienie hetmana w niewłaściwe miejsce), jak było w algorytmie standard backtracking, a brak możliwości wstawienia wartości kolejnej zmiennej, przez co ogranicza się liczbę przejrzanych przypadków



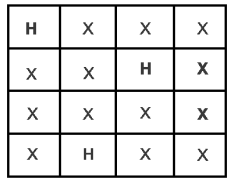
Rysunek 5.2. Ustawienie drugiego hetmana.

przyspieszając znacznie proces przeszukiwania.
W kolejnym kroku podstawia się nową wartość za zmienną K2 (druga i ostatnia wartość w domenie): [1,4,K3,K4]. Po kolejnym uaktualnieniu domen szachownica wygląda jak na rys. 5.3:
Tym razem jest możliwość ukonkretnienia zmiennej K3 wartością 2. Lista



Rysunek 5.3. Ponowne ustawienie drugiego hetmana.

zmiennych wyglądać wtedy będzie tak: [1,4,3,K4] (szachownica po aktualizacji domen jak na rys. 5.4).
Niestety po ustawieniu trzeciego hetmana domena zmiennej K4 pozosta-



Rysunek 5.4. Ustawienie trzeciego hetmana.

ła pusta, w związku z czym algorytm musi dokonać nawrotu i spróbować znaleźć inną wartość dla zmiennej K3. Ponieważ nie jest to możliwe (nie ma więcej wartości w domenie zmiennej K3) algorytm dokonuje kolejnego nawrotu próbując znaleźć inną wartość dla zmiennej K2, co również kończy się niepowodzeniem. Dopiero kolejny nawrót do zmiennej K1 pozwala na znalezienie kolejnej wartości tej zmiennej: [2,K2,K3,K4]. Po aktualizacji domen szachownica jak na rys 5.5.
Jest tylko jedna wartość jaką można w tej sytuacji podstawić pod zmienną

x	x		
H	x	x	x
x	x		
x		x	

Rysunek 5.5. Ponowne ustawienie pierwszego hetmana.

K2: [2,4,K3,K4]. Po kolejnej aktualizacji domen szachownica jak na rys. 5.6. Znów po aktualizacji domen została tylko jedna wartość, którą można pod-

x	x		
H	x	x	x
x	x	x	
x	H	x	x

Rysunek 5.6. Ponowne ustawienie drugiego hetmana.

stawić za zmienną K3: [2,4,1,K4]. Kolejna aktualizacja domen (rys. 5.7). W tym punkcie widać, iż została jedna wartość w domenie zmiennej K4:

x	x	H	x
H	x	x	x
x	x	x	
x	H	x	x

Rysunek 5.7. Ustawienie trzeciego hetmana.

[2,4,1,3] (Szachownica na rys. 5.8).

W ten sposób udało się znaleźć pierwsze rozwiązanie problemu ustawienia 4 hetmanów na szachownicy 4X4. Algorytm jednak nie poprzestaje na znalezieniu jednego rozwiązania i próbuje znaleźć następne: następują kolejne nawroty i podstawianie dalszych wartości z domen. Algorytm kończy działanie, gdy nie będzie miał więcej możliwości podstawień.

Algorytm Forward checking może mieć zaimplementowane udogodnienie polegające na tym, że sprawdzana jest nie tylko najbliższa domena ale także zaznacza się takie wartości, które na pewno po ukonkretnieniu w następnym kroku będą odrzucone. W przypadku problemu hetmanów chodzi tu o sąsiadujące ze sobą w linii, bądź na ukos pary pozycji na szachownicy, jeśli postawi się na którejś z nich hetmana, to na pewno nie będzie można go

X	X	H	X
H	X	X	X
X	X	X	H
X	H	X	X

Rysunek 5.8. Pierwsze rozwiązanie problemu.

postawić na pozycji sąsiedniej. Taki zmodyfikowany algorytm nosi nazwę: forward checking + looking ahead.

5.4. Ciekawe predykaty

Język ECLiPSe ma bardzo rozbudowany zestaw predykatów standardowych ułatwiających realizację zadań przeszukiwania. Poniżej zostanie przedstawione kilka najciekawszych z nich.

5.4.1. Predykat *cumulative*

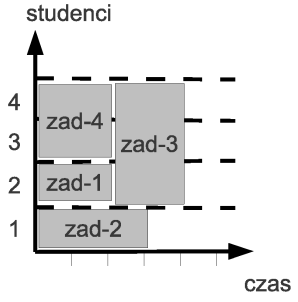
Jednym z popularnych problemów opierających się na przeszukiwaniu i optymalizacji jest problem harmonogramowania. Na czym ów problem polega?
Polega on na odpowiednim dopasowaniu istniejących zasobów (ludzkich, maszyn itp.), zadań do realizacji, czasochłonności poszczególnych zadań i punktu ich rozpoczęcia. Zazwyczaj wiąże się to z maksymalizacją wykorzystania dysponowanymi zasobami lub minimalizacją przerw. Najłatwiej chyba wyobrazić sobie problem harmonogramowania na przykładzie układania planu zajęć: prowadzący to zasoby, zadania to zajęcia do przeprowadzenia, czasochłonność to liczba godzin poszczególnych zajęć a punkty rozpoczęcia to oczywiście godziny rozpoczęcia zajęć. Pełna realizacja takiego programu (z uwzględnieniem sal, preferencji prowadzących itp.) jest bardzo skomplikowana, ale w najprostszej wersji problem ułożenia poszczególnych zajęć można zamodelować korzystając z predykatu wbudowanego *cumulative*. Jest kilka jego wersji (w zależności od liczby argumentów), gdzie najprostsza ma taką postać:
cumulative([Lista czasów startu], [lista czasów trwania], [lista zużycia zasobów poszczególnych zadań], zasoby).
Można teraz sobie wyobrazić pracę zaliczeniową dla grupy czterech studentów. Wiadomo, że do jego wykonania potrzebne jest zrobienie 4 różnych podzadań, z których każde wymaga do realizacji odpowiedniej liczby ludzi (niektóre muszą być wykonywane przez tylko jednego studenta, inne wyma-

gają pracy zespołowej) i odpowiedniego czasu. Problem polega na znalezieniu najlepszego podziału zadań. Lista zadań:
— zadanie 1 – 1 osoba 2 dni,
— zadanie 2 – 1 osoba 3 dni,
— zadanie 3 – 3 osoby 2 dni,
— zadanie 4 – 2 osoby 2 dni.

Oczywiście kryterium „najlepszości" podziału jest nieprecyzyjne ale na tym etapie jeszcze nie będzie analizowane, choć niewątpliwie dla studentów będzie ważne który z nich i kiedy będzie realizował dane zadanie. W pierwszym kroku należy znaleźć wszystkie możliwe podziały zadań dla poszczególnych studentów, z których dopiero będą wybierane te najlepsze. Właśnie do znalezienia wszystkich ustawień prac wykonywanych przez studentów można wykorzystać predykat *cumulative*. Jak to zrobić? Przede wszystkim należy zastanowić się czego się tak na prawdę szuka. Ponieważ nie ma teraz znaczenia jak się owi studenci nazywają i który konkretnie będzie realizował które zadanie można ich po prostu ponumerować. To co jest ważne i czego się powinno w tym miejscu szukać to czasy rozpoczęcia poszczególnych czynności. Dlaczego? Ano dlatego, że jeśli głównym celem jest znalezienie jak najlepszego podziału to czas wykonania całej pracy powinien mieć tu kluczowe znaczenie. Jak w takim razie powinien być zamodelowane wszystkie ograniczenia związane z opisem tej sytuacji? Gdy chcieć to wyrazić tylko za pomocą języka PROLOG była by to grupa dość skomplikowanych klauzul, w przypadku języka ECLiPSe wystarczy posłużenie się jednym predykatem cumulative:

```
cumulative([S1,S2,S3,S4],[2,3,2,2],[1,1,3,2],4)
```

Oczywiście sam powyższy predykat zadania nie rozwiązuje, trzeba jeszcze przygotować całą „otoczkę" programu, określić domenę, prezentację wyników itp. Przykładowy podział zadań wyrażony w formie wykresu Gantta może wyglądać jak na rys. Realizacja zadań bardziej skomplikowanych wy-



Rysunek 5.9. Podział zadań.

maga oczywiście znacznie bardziej rozbudowanego programu, gdyż zasobów może być więcej i mogą być ze sobą powiązane na wiele różnych sposobów. Oprócz szkolnych zastosowań takich jak układanie planu zajęć, problem harmonogramowania jest obecny przy praktycznie każdym planowaniu, wszędzie tam, gdzie zasoby są ograniczone i muszą być wykorzystane do realizacji szeregu zadań. Analizowany problem ma jeszcze jeden nie omówiony a bardzo ważny składnik: znalezienie nawet bardzo dużej liczby różnych podziałów zadań nie rozwiązuje problemu. Problem będzie rozwiązany, gdy znajdzie się rozwiązanie najlepsze.

5.4.2. Optymalizacja

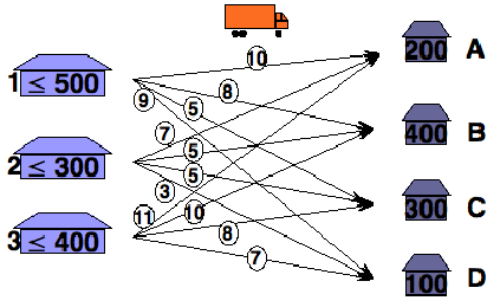
Które opcja jest najlepsza? Takie pytanie zadaje sobie większość ludzi w większości sytuacji decyzyjnych i w większości sytuacji człowiek intuicyjnie znajduje najlepsze, albo przynajmniej wystarczające rozwiązanie. Niestety w przypadku wielu rzeczywistych problemów biznesowych taka „ludzka" metoda zawodzi. Zazwyczaj jest zbyt wiele, zbyt skomplikowanych możliwości by można było wygenerować chociażby zadowalające rozwiązanie. I co wtedy? Wtedy oczywiście można wykorzystać któryś z języków CLP aby przeszukał wszystkie możliwości i znalazł te najlepsze. Oczywiście w tym punkcie pojawia się postawione już wcześniej pytanie: a po czym poznać najlepsze rozwiązanie?
Zazwyczaj człowiek kieruje się wieloma, często bardzo różnymi względami „najlepszości", które niejednokrotnie są ze sobą nieporównywalne i często niemierzalne, bądź trudnomierzalne. Chcąc szukać rozwiązania optymalnego za pomocą komputera trzeba je jednak w jakiś sposób sformalizować, dlatego musi być ono precyzyjnie zdefiniowane i oczywiście mierzalne. W większości zadań optymalizacyjnych pracę rozpoczyna się od zdefiniowania tzw. funkcji celu (zwanej czasem wskaźnikiem jakości). Czym jest funkcja celu? Jest formalne wyrażenie zależności, która dla optymalnego rozwiązania powinna być zmaksymalizowana lub zminimalizowana. Funkcja celu może maksymalizować zysk albo minimalizować koszty, minimalizować czas potrzebny na realizację zadań itp. Jednym z najpopularniejszych mechanizmów optymalizacyjnych wykorzystywanych w językach CLP jest mechanizm branch and bound. Nie jest to w pełni algorytm a raczej strategia poszukiwania najlepszego rozwiązania, która działa w połączeniu z algorytmem przeszukiwania (np. forward checking) [17]. Jak działa ten mechanizm? Jeśli celem jego działania jest minimalizacja wartości funkcji celu to w trakcie przeszukiwania, dla pierwszego znalezionego rozwiązania spełniającego ograniczenia wyznaczana i zapamiętywana jako najlepsza jest wartość funkcji celu. Dla każdego kolejnego rozwiązania również wyznacza się wartość funkcji celu i

gdy nowa wartość jest mniejsza od dotychczas zapamiętanej, to ona zajmuje jej miejsce (oczywiście wraz ze zbiorem wartości zmiennych decyzyjnych). Jeśli zaś nowa wartość nie jest mniejsza od wcześniej znalezionej, to program nic nie zmienia i dalej poszukuje kolejnych rozwiązań. Mechanizm kończy pracę, gdy sprawdzi wszystkie rozwiązania.

W języku ECLiPSe jest biblioteka *branch_and_bound*, którą można wykorzystać we własnych programach. Do wykorzystywania metody branch and bound służy przede wszystkim predykat *bb_min(Cel, Funkcja celu, Opcje)*, gdzie:

— Cel to predykat zależny od zmiennych decyzyjnych.
— Funkcja celu, to predykat opisujący wskaźnik jakości.
— Opcje – tu można opisać różne parametry procesu znajdowania najlepszych rozwiązań, m.in. to, czy po znalezieniu pierwszego rozwiązania szukać dalej (standardowa opcja), czy rozpoczynać przeszukiwanie od nowa, można wstawić tu górne, lub dolne wartości funkcji celu, minimalną oczekiwaną poprawę funkcji celu itp.

W bibliotece *branch_and_bound* alternatywnym dla predykatu *bb_min* jest predykat *minimize(Cel, funkcja celu)*, który odpowiada predykatowi *bb_min* z domyślnymi opcjami. Poniżej zostanie przedstawiony przykład zadania optymalizacyjnego z wykorzystaniem tego predykatu (zadanie pochodzi z materiałów ze strony www.eclipse.cp.org):



Rysunek 5.10. Zagadnienie transportowe, źródło: www.eclipseclp.org.

Trzy fabryki produkują jeden towar. Towar ten jest odbierany przez cztery hurtownie. Ponieważ tak fabryki jak i hurtownie nie są zgrupowane razem, w związku z czym między nimi są różne odległości i, co za tym idzie, są różne koszty transportu. Co więcej każda z fabryk ma inną zdolność

produkcyjną (czyli może wyprodukować nie więcej niż wynosi jej zdolność, ale może mniej), natomiast każda z hurtowni wymaga konkretnej ilości towaru (ani mniej ani więcej). Jest to klasyczny przykład tzw. zagadnienia transportowego Całe zadanie przedstawione jest na rys. 5.2. Podane tam są moce produkcyjne fabryk, zapotrzebowanie hurtowni oraz jednostkowe koszty transportu z każdej fabryki do każdej hurtowni.

Celem zadania jest znalezienie najtańszej opcji przewozu towaru z fabryk do hurtowni. Poniżej jest przedstawiony program będący z jednej strony modelem problemu, z drugiej zaś jego rozwiązaniem.

Listing 5.2. Zadanie transportowe

```
1 :- lib(ic).
  :- lib(branch_and_bound).
3 solve(Vars, Cost) :-
      model(Vars, Obj),
5     Cost #= eval(Obj),
      minimize(search(Vars, 0, first_fail,
7     indomain_split, complete, []), Cost).
  model(Vars, Obj) :-
9     Vars = [A1, A2, A3, B1, B2, B3, C1, C2, C3, D1, D2, D3],
      Vars :: 0..inf,
11    A1 + A2 + A3 $= 200,
      B1 + B2 + B3 $= 400,
13    C1 + C2 + C3 $= 300,
      D1 + D2 + D3 $= 100,
15    A1 + B1 + C1 + D1 $=< 500,
      A2 + B2 + C2 + D2 $=< 300,
17    A3 + B3 + C3 + D3 $=< 400,
      Obj =
19        10*A1 + 7*A2 + 11*A3 +
          8*B1 + 5*B2 + 10*B3 +
21        5*C1 + 5*C2 +  8*C3 +
          9*D1 + 3*D2 +  7*D3.
23 $
```

Program rozpoczyna się od poleceń dołączenia bibliotek (*ic* oraz *branch_and_bound*. Analizę głównej części programu najlepiej jest rozpocząć od przyjrzenia się predykatowi *model*, który definiuje ograniczenia, oraz funkcję celu. Predykat *model(Vars, Obj)*. Ma dwa argumenty: pierwszy z nich Vars jest listą zmiennych oznaczających ilość sztuk towaru przewiezionych z fabryki do hurtowni (wiersz 9 deklaruje, że zmienna Vars to lista zmiennych, przy czym, np. zmienna A1 oznacza ilość sztuk towaru przewiezioną z fabryki 1 do hurtowni A). W wierszu 9 deklarowana jest domena zmiennych: od 0 do nieskończoności. Kolejne wiersze, od 11 do 17 definiują ograniczenia,

związane z zapotrzebowaniem (suma liczby towarów przywieziona do każdej hurtowni musi być równa jej zapotrzebowaniu) i mocami produkcyjnymi fabryk (suma towarów wywiezionych z każdej fabryki musi być mniejsza lub równa mocom produkcyjnym poszczególnych fabryk).
Zmienna Obj jest równa łącznemu kosztowi transportu towaru (suma iloczynów ilości towaru i jednostkowych kosztów) i będzie później wykorzystana jako funkcja celu.
Drugi (a właściwie pierwszy) predykat, czyli *solve(Vars, Cost)* modeluje sam proces przeszukiwania oraz optymalizację, czyli szukanie najlepszego rozwiązania.
Aby dokładniej wyjaśnić proces optymalizacji należy przyjrzeć się jeszcze dwom predykatom standardowym wykorzystanym w programie. predykat *eval* „wylicza" wartość zmiennej, w tym przypadku ze względu na wykorzystanie znaku # jest on traktowany jako ograniczenie całkowitoliczbowe.
Predykat *search* jest uogólnieniem predykatu *labelling* i jego zadaniem jest uruchomienie procesu przeszukiwania (ukonkretniania zmiennych. Ma sześć argumentów: *search(Lista, Arg, Wybieranie zmiennych, Wybieranie wartości, Metoda, Opcje)*, gdzie [17]:

— Lista, to lista ukonkretnianych zmiennych lub termów.
— Arg, to oznaczenie, czy Lista jest listą tylko zmiennych (wartość 0), czy listą termów (wtedy Arg pokazuje wybrany argument termu).
— Wybieranie zmiennych, to heurystyka z jaką mechanizm wnioskowania dobiera zmienne do ukonkretnienia. Zastosowana tu wartość *first_fail* oznacza, że jako pierwsze będą brane pod uwagę zmienne z najmniejszą dziedziną.
— Wybieranie wartości, to znów strategia wyboru wartości podczas ukonkretniania zmiennych. W programie wybrano *indomain_split*, która dzieli domenę na pół, połówka, w której będzie porażka zostanie usunięta, reszta zaś będzie dalej połowiona.
— Metoda, to jedna z metod przeszukiwań, tu wykorzystano *complete*, w którym testowane są wszystkie ukonkretnienia zmiennych.
— Opcje, tu można wymienić dodatkowe opcje predykatu (np. zwracanie liczby nawrotów, można ustalić ilość odwiedzonych węzłów drzewa poszukiwań, itp.). W programie nie wykorzystano żadnych dodatkowych opcji.

Predykat *solve* będzie spełniony, gdy spełniony będzie predykat *model*(wiersz numer 4), wartość zmiennej Cost będzie ukonkretnioną i wyliczoną wartością funkcji celu (wiersz 5) i system znajdzie najmniejszą wartość przeszukując wg opisanych wyżej opcji i funkcji celu opisanej przez zmienną Cost (wiersze 6 do 7).
Wywołanie, po kompilacji, predykatu *solve(Vars, Cost)* spowoduje wypisanie na ekranie zawartości zmiennej Vars, czyli listy zmiennych opisujących

ilość sztuk towaru przewiezioną z każdej z fabryk do każdej hurtowni. Wartość zmiennej Cost będzie oznaczała łączny koszt transportu.
Problemy optymalizacyjne można rozwiązywać nie tylko za pomocą mechanizmu branch and bound. Język ECLiPSe oferuje również solvery, które można dołączać do programu w postaci bibliotek (takim solwerem jest np. biblioteka *eplex*), pozwalają one rozwiązywać zadania nie tylko dla problemów dyskretnych ale także liniowych, symbolicznych itp. Jak już wcześniej było wyjaśnione, pozycja ta z wielu względów nie ma ambicji być kompletnym kursem języka CLP, ale zainteresowanym czytelnikom można zdecydowanie polecić dość przystępnie napisaną książkę [17], bądź materiały ze strony internetowej producenta języka ECLiPSe (www.eclipseclp.org).

Rozdział 6

Teoria gier

Wiele sytuacji życiowych, biznesowych, technicznych polega na rozwiązywaniu różnego rodzaju konfliktów i bardzo często zdarza się podczas podejmowania decyzji, że trzeba przewidzieć jak zachowa się konkurent. Konkurenci mają zazwyczaj swoje cele, które często nie są zbieżne ze sobą, dlatego trzeba brać pod uwagę możliwości postępowania każdego z nich zakładając, że zachowuje się racjonalnie i dąży do realizacji swoich celów.

Problematyką optymalnego zachowania w sytuacjach konfliktowych zajmuje się specjalny dział matematyki zwany teorią gier. Jest on szczególnie często łączony z problemami ekonomicznymi i biznesowymi, gdyż najczęściej tam znajduje zastosowanie. Ponieważ podejmowanie różnego rodzaju decyzji, także ekonomicznych, jest w zakresie zainteresowań sztucznej inteligencji, w związku z czym rezultaty badań w dziedzinie teorii gier znajdują zastosowanie w narzędziach sztucznej inteligencji.
Pomimo tego tematyka teorii gier rzadko jest prezentowana w literaturze poświęconej sztucznej inteligencji, choć wdrożenia różnego rodzaju systemów zawierających jej mechanizmy zdarzają się wcale nie aż tak rzadko. Dla czytelników chcących w miarę „bezboleśnie" wdrożyć się w tematykę teorii gier warto polecić ostatnią powieść Stanisława Lema: „Fiasko", w której teoria gier (i komputer nią się posługujący) odgrywa właśnie niepoślednią rolę.

6.1. Dlaczego gry?

Na wstępie musi się pojawić pytanie podstawowe: dlaczego teoria gier? Co to za gry i o co w nich chodzi? Sytuacje konfliktowe między jakimiś konkurentami można przedstawić właśnie jako jaką grę: grę w której są gracze, jakieś możliwości postępowania i strategie, oraz rezultaty gry związane z jakimiś przegranymi lub wygranymi. Warto w związku z tym nieco dokładniej przyjrzeć się takiej grze i jej elementom. Każda gra powinna mieć [12]:

— Wyszczególnionych uczestników gry,
— Wyszczególnione możliwości postępowania każdego gracza,
— Zadeklarowaną wiedzę każdego gracza odnośnie gry, gry jej uczestników i potencjalnych wygranych i przegranych,
— Wyszczególnione cele, wygrane i przegrane poszczególnych graczy.

Powyższe punkty to oczywiście cechy modelu. W rzeczywistości większość sytuacji jest znacznie bardziej skomplikowana i chcąc je zamodelować jako gry trzeba poczynić pewne upraszczające założenia. Nie ma oczywiście w tym nic złego, w większości sytuacji do lepszego ich rozumienia i rozwiązania stosuje się różnego rodzaju uproszczenia. Jakiego rodzaju są to uproszcze-

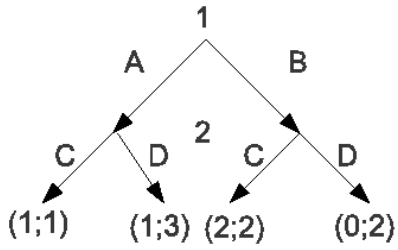
nia? Liczba uczestników gry może być np. nieznana – tak jest w sytuacji modelowania różnego rodzaju aukcji i gry rynkowych, nikt nie może zagwarantować, że nie pojawią się nowi gracze na rynku. Czasem jeden gracz może się podzielić na dwóch itp. Zazwyczaj w takim razie zakłada się uproszczenie gry do kilku głównych agentów. Podobnie nie zawsze znane są wszystkie możliwości działań poszczególnych graczy, bądź jest ich bardzo wiele, wtedy również czasem można pozwolić pewne uproszczenie polegające na zawężeniu się do kilku najważniejszych, bądź najbardziej prawdopodobnych akcji. Nie zawsze również wiadomo jaką wiedzę mają poszczególni agenci: czy wiedzą jakie będą rezultaty ich działań, jakie będą rezultaty działań konkurencji, mogą nie wiedzieć jakie są możliwości działań konkurencji, ani nawet kto jest ich konkurentem w grze. Mogą być też sytuacje, w których poszczególni gracze nie mają wiedzy ale mają jakieś przypuszczenia itp. Odwzorowanie tego typu sytuacji w matematycznym modelu jest zazwyczaj bardzo trudne i często zakłada się tu pewne uproszczenia. Wreszcie, czasem, nie są znane, bądź są bardzo trudne do ustalenia rezultaty gry: czy są jakieś wygrane?, czy wygrane są mierzalne?, czy wygrane (bądź przegrane) poszczególnych graczy są porównywalne ze sobą? (np. celem jednego gracza jest zysk a drugiego prestiż - jak to porównać?), jeden gracz może mieć kilka różnych celów i czy nie są one sprzeczne? itp. Tego rodzaju wątpliwości jest bardzo wiele i zawsze w tego typu sytuacjach stosuje się różne uproszczenia. Ważne w nich jest to aby upraszczały kwestie mało istotne a koncentrowały się na sprawach ważnych, by oddawały jak najprawdziwszy obraz konfliktu i pozwalały go właściwie rozwiązać.

6.2. Najprostsza gra

Jak w takim razie przedstawić jakąś przykładową prostą sytuację konfliktową w postaci gry? Najprostszym modelem gry jest drzewo i jako drzewo można przedstawić sobie przykładową grę: Jest dwóch graczy i pierwszy z nich ma dwie możliwości działania (akcja A i akcja B), drugi gracz może każdą akcję gracza pierwszego również odpowiedzieć dwoma akcjami (akcją C i akcją D). Rezultaty gry są takie:

— Jeśli gracz pierwszy wybierze A a gracz drugi C to gracz pierwszy zarabia 1 i gracz drugi zarabia również 1,
— Jeśli gracz pierwszy wybierze A a gracz drugi D to gracz pierwszy zarabia 1 i gracz drugi zarabia 3,
— Jeśli gracz pierwszy wybierze B a gracz drugi C to gracz pierwszy zarabia 2 i gracz drugi zarabia również 2,
— Jeśli gracz pierwszy wybierze B a gracz drugi D to gracz pierwszy zarabia 0 i gracz drugi zarabia 2.

Taką grę można przedstawić w formie drzewa decyzyjnego i jest to najprostsza forma reprezentacji gry (rys 6.1) W nawiasach podano wyniki: pierw-



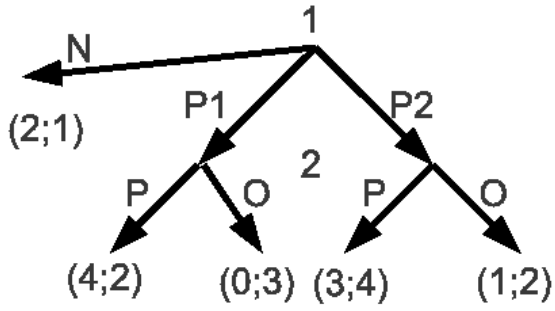
Rysunek 6.1. Przykładowa gra w postaci drzewa, źródło [12].

sza pozycja wygrana gracza pierwszego, druga wygrana gracza drugiego). Analizując powyższą grę warto się zastanowić jak ją rozwiązać, czyli jak poszczególni gracze powinni się racjonalnie zachować, czyli tak by zmaksymalizować swoją wygraną i, ewentualnie, zminimalizować przegraną.
Pierwszą rzeczą na którą warto tu zwrócić uwagę jest wiedza graczy, tzn. co który z nich wie. Jeśli założy się, że wiedzą wszystko o grze, czyli to jakie są możliwości działania własne i konkurenta i jakie one będą miały skutki to można sobie intuicyjnie przedstawić tok rozumowania gracza pierwszego, który może myśleć tak: „Jeśli wybiorę A, to niezależnie od tego co wybierze gracz drugi zawsze wygram 1, jeśli zaś wybiorę B, to mogę wygrać 2 a mogę nie wygrać nic, wygram 2 wtedy, gdy gracz drugi wybierze opcję C, nic nie wygram jeśli wybierze on D. Ponieważ nie ma on żadnego interesu w tym, żeby wybierał opcję D, bo zawsze dostanie tą samą wygraną, więc prawdopodobnie wybierze C i ja wtedy też wygram 2.” Może jednak gracz pierwszy co nieco powątpiewać w gracza drugiego i wybrać „bezpieczną” opcję wyboru A, gdzie na pewno wygra 1.
Gdyby jednak nieco zmienić grę i wygrana w sytuacji, gdy gracz pierwszy wybrał B a gracz drugi D wynosiła by 0 dla gracza pierwszego a 3 dla gracza drugiego, wtedy rozsądek gracza pierwszego jednoznacznie już narzuciłby wybór opcji A (przy opcji B gracz drugi na pewno zdecydował by się na opcję D ze względu na wyższą wygraną i A dostałby wtedy 0).
Przy tego rodzaju grach problemem może być trudna do ogarnięcia wielkość drzew, gdyż gra może mieć znacznie więcej aktorów, każdy aktor może mieć bardzo wiele możliwości działania (niejednokrotnie może mieć teoretycznie nieskończoną liczbę możliwych działań) i etapów gry również może być znacznie więcej. Bez trudu można sobie wyobrazić zamodelowanie internetowej aukcji jako gry. Problem w jej reprezentacji leży w ogromnej liczbie potencjalnych uczestników (wszyscy zarejestrowani), ogromne moż-

liwości działań (pasowanie, czekanie, podbijanie o 1 zł. o 2 zł itd.) i bardzo duża ilość „etapów" gry związanych z możliwością wielokrotnego podbijania ceny. Gdyby spróbować narysować takie drzewo gry byłoby ono ogromne i absolutnie nieczytelne. Warto przy tym zwrócić jednak uwagę, że można poczynić pewne upraszczające założenia, które pozwolą na stworzenie uproszczonego ale czytelnego drzewa, które pozwoli ową grę rozwiązać (można tak zrobić np. usuwając najbardziej nieprawdopodobne rozwiązania i sytuacje).

6.3. Wybór inwestycji

Aby łagodnie przejść od najprostszej gry do problemów decyzyjnych zostanie poniżej przedstawiony przykład dylematu jaki może mieć hipotetyczny przedsiębiorca. Problem wygląda tak:
Pewien przedsiębiorca ma znakomity pomysł na rozwinięcie swojej działalności, pomysł ma dwa różne warianty i ma dylemat, gdyż gdyby zaprosił do udziału w przedsięwzięciu konkurenta to obaj mieliby większe szanse na odniesienie sukcesu. Niestety konkurent może propozycję odrzucić, wykorzystując jednocześnie pomysł do zwiększenia swoich zysków. Jakie w takim razie są możliwości działania poszczególnych przedsiębiorców? Przedsiębiorca pierwszy ma możliwość zaproponowania konkurentowi wariantu I współpracy, wariantu II współpracy, bądź nie proponowania współpracy w ogóle. Przedsiębiorca drugi ma cztery strategie działania: może założyć, że przyjmie wszystkie propozycje współpracy, może założyć, że przyjmie wariant pierwszy i odrzuci wariant drugi, może założyć, że odrzuci wariant pierwszy i przyjmie wariant drugi i może wreszcie odrzucić wszystkie potencjalne propozycje. Na rys 6.2 jest zaprezentowane drzewo gry. P1 i P2



Rysunek 6.2. Drzewo gry „wybór inwestycji", źródło [12].

oznaczają propozycje wariantu pierwszego i drugiego, N oznacza nie składa-

nie propozycji w ogóle. Dla gracza drugiego P oznacza przyjęcie propozycji, O odrzucenie propozycji.
Grę tą można przedstawić także w innej formie. Na wstępie trzeba zwrócić uwagę, że gracz pierwszy ma trzy strategie:
— Strategia I: nie składać propozycji,
— Strategia II :złożyć propozycję P1,
— Strategia III: złożyć propozycję P2.
Gracz drugi ma cztery możliwe strategie:
— Strategia I: zgadzać się na wszystkie propozycje,
— Strategia II: zgodzić się na propozycję P1 i odrzucić propozycję P2,
— Strategia III: zgodzić się na propozycję P2 i odrzucić propozycję P1,
— Strategia IV: odrzucić wszystkie propozycje.
Poszczególne strategie wraz z wygranymi poszczególnych graczy można przedstawić w formie tabeli: Jak taką grę w takim razie rozwiązać i jak powinni

Tabela 6.1. Macierz gry

1/2	I	II	III	IV
I	(2;1)	(2;1)	(2;1)	(2;1)
II	(4;2)	(4;2)	(0;3)	(0;3)
III	(3;4)	(1;2)	(3;4)	(1;2)

postępować obaj gracze? Co powinni wybrać? Gracz pierwszy ma do wyboru trzy strategie i nie wie co wybierze konkurent, wie jednak, że będzie on wybierał najracjonalniej, czyli tak by zarobić jak najwięcej. Wybór nie proponowania żadnej współpracy daje pewny wynik i wypłatę 2. Złożenie propozycji pierwszej daje możliwość zarobienia 4 ale tylko wtedy, gdy konkurent pójdzie na współpracę. Wiadomo jednak, że konkurent ów więcej zarobi wtedy, gdy odrzuci współpracę, więc raczej na pewno wybierze tą opcję, w związku z czym gracz pierwszy nic wtedy nie zarobi. Gdyby gracz pierwszy złożył drugą propozycję to może zarobić 3, gdy będzie przyjęta, bądź tylko 2, gdy będzie odrzucona. Wiadomo jednak, że przyjęcie propozycji jest w tym wypadku korzystne dla gracza drugiego, bo zarobi więcej, niż ją odrzucając. W takiej sytuacji gracz pierwszy składa propozycję P2 a gracz drugi ją przyjmuje i obaj zarabiają (gracz pierwszy 3 a gracz drugi 4).
Przyglądając się strategiom obu graczy nieco bardziej formalnie można przeanalizować je niejako „wstecz” od decyzji gracza drugiego do decyzji gracza pierwszego: Gracz drugi ma cztery możliwe strategie. Niewątpliwie nie opłaca się przyjmować propozycji pierwszej (P1), gdyż zarobi więcej ją odrzucając, natomiast opłaca się graczowi drugiemu przyjąć propozycję drugą (P2), gdyż wtedy zarobi więcej niż ją odrzucając. W takiej sytuacji optymalną strategią gracza drugiego jest strategia III.

Gracz pierwszy powinien przeprowadzić analizę postępowania dla gracza drugiego aby poznać jego preferencje a następnie wybrać taka strategię aby, w ramach wyboru gracza drugiego, zarobić najwięcej. W tej sytuacji najlepszym wyjściem jest oczywiście złożenie propozycji P2.
Opisany wyżej mechanizm analizy gry „od końca" w teorii gier nosi nazwę indukcji wstecznej. Jej wynikiem są strategie, które są najlepszymi odpowiedziami na siebie. Takie pary strategii noszą nazwę równowag Nasha.
Opisany wyżej algorytm szukania takiej równowagi sprowadza się do wyboru strategii maksymalizujących zyski (lub minimalizujących straty) dla poszczególnych graczy, analizując ich wybory od ostatniego ruchu.

6.4. Strategia mieszana

Niestety są takie gry, w których nie zawsze możliwe jest wyznaczenie równowagi za pomocą indukcji wstecznej. Można sobie wyobrazić grę, w której jest dwóch uczestników i każdy z nich (bez wiedzy drugiego) wybiera jedną stronę monety. Jeśli obaj wybrali tą samą wygrywa gracz pierwszy, jeśli różne wygrywa gracz drugi. Jest to dość ciekawa gra, różna od tych prezentowanych wcześniej. Przede wszystkim jest to gra o niepełnej informacji, tzn. gracze nie wiedzą jaki ruch wykonał konkurent, poza tym jest to gra o sumie zero, czyli wygrana jednego gracza jest przegraną drugiego. Macierz tej gry jest zaprezentowana w poniższej tabelce:

Tabela 6.2. Macierz gry orzeł-reszka

1/2	O	R
O	(1;-1)	(-1;1)
R	(-1;1)	(1;-1)

Łatwo zauważyć, że w tego rodzaju grze nie ma możliwości znalezienia rozwiązania drogą indukcji wstecznej. W takich sytuacjach stosuje się tzw. strategie mieszane oparte na losowości. Polegają one na tym, że gracz przed podjęciem decyzji przeprowadza losowanie, na skutek którego z prawdopodobieństwem p1 wybiera orła, natomiast z prawdopodobieństwem $q1 = (1-p1)$ wybiera reszkę. Gdyby wtedy gracz drugi wybrał orła to oczekiwana wygrana wyniosłaby [12]:
$p1 \cdot 1 + q1 \cdot (-1) = 2p1 - 1$
Gdyby zaś wybrał reszkę to oczekiwana wygrana wyniosłaby:
$p1 \cdot (-1) + q1 \cdot 1 = 1 - 2p1$
W związku z tym jeśli prawdopodobieństwo p1 = 0,5, to niezależnie od wyboru gracza drugiego oczekiwana wygrana wyniesie 0. Gdyby wartości wygranych nie były symetryczne dla obu graczy (np. za układ orzeł-orzeł

wypłaty byłyby (2;-2) wtedy wyznaczenie wartości oczekiwanych mogło pomóc opracować właściwą strategię losowania p1 (prawdopodobieństwo to wcale nie musi wtedy być równe 0,5).
Warto jeszcze zwrócić uwag,ę, że w przypadku gier o sumie zero wystarczy podawać wartości tylko wygranej jednego gracza, która jest jednocześnie stratą gracza drugiego i odwrotnie: strata gracza pierwszego jest wygraną drugiego.

6.5. Dylemat więźnia

Jedną z najpopularniejszych sytuacji modelowanych za pomocą teorii gier jest tzw. dylemat więźnia. Na czym on polega?
Policja w czasie rutynowej kontroli złapała dwóch dżentelmenów w kradzionym samochodzie marki BMW. W bagażniku samochodu znaleziono łup z napadu na bank. Obu oskarżonych rozdzielono i trzymani są w osobnych celach bez możliwości porozumiewania się. Każdy z nich ma dwie możliwości postępowania: może się przyznać do napadu na bank, bądź nie. Jeśli obaj się nie przyznają to odpowiedzą tylko za kradzież samochodu (nie ma dowodu na to, że brali udział w napadzie) i zostaną skazani na rok więzienia, jeśli jeden się przyzna to będzie świadkiem koronnym i zostanie wypuszczony a drugi zostanie skazany na 6 lat „odsiadki". Jeśli obaj się przyznają to zostaną skazani na 4 lata więzienia. Co w takiej sytuacji powinni robić?
Na wstępie warto rozrysować macierz gry. Wygrane poszczególnych graczy oznaczać będą lata na wolności w ciągu najbliższych 6 lat:

Tabela 6.3. Macierz dylematu więźnia

1/2	przyznaje się	nie przyznaje się
przyznaje się	(2;2)	(6;0)
nie przyznaje się	(0;6)	(5;5)

W klasycznym dylemacie więźnia strategią dominującą jest przyznawanie się do dokonania włamania na bank, choć oczywiście najlepiej by wyszli na solidarnym nie przyznawaniu się do napadu. Przyczyna takiej sytuacji leży w braku wiedzy i zaufania co do poczynań wspólnika. Tego rodzaju sytuację można odnieść do zawiłości gier politycznych, biologii itp.
Oczywiście dylemat więźnia dotyczy nie tylko takiego konkretnego przypadku wypłat wygranych, odnosi się on generalnie do sytuacji, w których zachowane są stosunki większy-mniejszy między poszczególnymi wygranymi.
Nieco ciekawszą odmianą dylematu więźnia jest tzw. iterowany dylemat, czyli gra, w której dylemat ów powtarza się cyklicznie. Każdy gracz przy

każdej iteracji musi podejmować decyzję, czy współpracuje, czy zdradza. Oczywiście po zdradzie jednego współgracza mało jest prawdopodobne, że ktoś mu zaufa, dlatego warto dobrze rozważyć, czy i jeśli tak, to kiedy powinno się zdradzić. Generalnie problem ten jest dość skomplikowany i mogą być tu wykorzystywane różne strategie. Najprostsza polega na współpracy do pewnego punktu, potem jednorazowy maksymalny zarobek a następnie już tylko zdradzanie. Podejście to bazuje na założeniu zmniejszania się wartości wygranej w czasie: lepiej mieć 100 zł dzisiaj niż za rok. Do zamodelowania tego rodzaju podejścia konieczne jest poczynienie pewnego założenia: należy ustalić wielkość stopy dyskontowej, czyli procent z jakim wygrana traci na wartości z cyklu na cykl. Zakładając, że współczynnik dyskontowy gracza wynosi δ to łączna wygrana uwzględniając stratę wartości przyszłych wygranych wyniesie:

$$Wygrana = W + W \cdot \delta + W \cdot \delta^2 + W \cdot \delta^3 + \ldots$$

gdzie W to wygrana jednostkowa.
Macierz iterowanego dylematu więźnia można uogólnić: Gdzie: $D < C < A < B$.

Tabela 6.4. Uogólniona macierz dylematu więźnia

1/2	przyznaje się	nie przyznaje się
przyznaje się	(A;A)	(D;B)
nie przyznaje się	(B;D)	(C;C)

Gdyby założyć, że gracz pierwszy wybierze strategię polegającą na współpracy do pewnego czasu T a następnie zdradę, to łączna wygrana wyniesie:

$$Wygrana = A + A \cdot \delta + \ldots A \cdot \delta^{T-1} + B \cdot \delta^T + C \cdot \delta^{T+1} \ldots$$

W takiej sytuacji od wielkości poszczególnych wygranych i współczynnika dyskontowego zależy wybór czasu zerwania współpracy. Gdy współczynnik dyskontowy będzie spełniał nierówność [12]:

$$\delta > \frac{B - A}{B - C}$$

Wtedy graczowi pierwszemu nie opłaca się zdradzać w ogóle i powinien odrzucić współpracę dopiero po zdradzie gracza drugiego. Warto zauważyć, że w takiej sytuacji równowaga w grze inaczej się kształtuje niż w grze jednorazowej, gdzie najlepszą strategią jest zdrada.
Doświadczenie pokazuje też, że w przypadku gier o bardzo długim czasie (w niektórych sytuacjach zakłada się czas nieskończony) korzystne jest wznowienie współpracy po pewnym czasie „ukarania" zdrajcy. Gdy gra ma

oznaczony i dość krótki czas (liczbę iteracji) to opłaca się zdradzić przed końcem gry (wysoka jednorazowa wygrana a konkurent nie ma możliwości „ukarać" gracza) problem oczywiście polega na tym, że konkurent też to wie i może (aby uniknąć straty) uprzedzić zdradę gracza. Cała gra w tym wypadku opiera się na problemie wyważenia między jak najdłuższą współpracą i wynikającym z niej zyskiem a koniecznością uprzedzenia zdrady konkurenta. Zilustrować można to sytuacją „z życia politycznego wziętą". Po wyborach parlamentarnych partie zazwyczaj bardzo chętnie nawiązują koalicję rządzącą, bo i władza, i ciepłe posadki dla krewnych i przyjaciół, i inne wynikające z nich korzyści, ale w trakcie upływu czasu i zbliżającego się terminu kolejnych wyborów coraz mniej chętnie tkwią w tej koalicji i najprawdopodobniej niedługo przed kolejnymi wyborami mniejszy koalicjant postanowi odejść „oburzony" oczywiście postawą większego koalicjanta zbijając sobie jednocześnie punkty u niezadowolonych wyborców. Podobnie „kombinuje" większy koalicjant zastanawiając się kiedy „wyrzucić" współkoalicjanta „zwalając" na niego winę za brak sukcesów. Przyglądając się historii działalności parlamentu można zauważyć, że zastanawiająco często koalicje rozpadały się właśnie w takich okolicznościach...

6.6. Racjonalność postępowania i wygrane

Niestety nie zawsze działania racjonalne dają najlepsze rezultaty dla wszystkich graczy, zilustrować można to popularnym przykładem [12]: We wsi jest pięciu rolników, z których każdy ma dwie krowy. Jest też pastwisko, na którym krowy te można paść za darmo: niestety owo pastwisko ma ograniczoną „wydajność" w związku z czym im więcej jest na nim krów tym mniej się najadają i bardziej trzeba je dożywiać w oborze. Każdy z graczy (rolników) ma trzy możliwości działania: wyprowadzić na pastwisko obie swoje krowy, tylko jedną, bądź nie wyprowadzać żadnej, tylko karmić ją w oborze. Poniżej zostanie przedstawiona macierz gry z punktu widzenia jednego gracza (dla wszystkich graczy musiałaby być macierz pięciowymiarowa). Wygrane graczy to pieniądze zaoszczędzone na tym, że krowa jadła na pastwisku i mniej zjadła w oborze. Najlepiej dla wszystkich rolników

Tabela 6.5. Macierz gry z krowami i pastwiskiem

liczba krów własnych / cudzych	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	11	10	9	8	7	6	5	4	3
2	20	18	16	14	12	10	8	6	4

byłoby, gdyby każdy z nich wypasał jedną krowę, wtedy każdy miałby taki

sam, optymalny dla gry zysk. Niestety żaden z rolników nie ma interesu w tym, żeby wypasać na pastwisku tylko jedną krowę, gdyż zawsze dodając drugą osiągnie lepszy wynik, niezależnie od tego, czy konkurencja wystawi jedną, czy dwie krowy. W związku z tym każdy gracz wystawi dwie krowy i zysk każdego z nich wyniesie tylko 4. Z tego przykładu widać wyraźnie, że w pewnych sytuacjach zachowanie racjonalne z punktu widzenia poszczególnych graczy przekłada się na gorszy wynik, niż w sytuacji współdziałania. Samo źródło tego problemu leży w ograniczoności wspólnych zasobów, z których korzystać powinni wszyscy gracze. Jak w takim razie można sobie z tego rodzaju problemem poradzić? Niestety konieczny będzie wpływ siły zewnętrznej, tzn. sołtys może np. narzucić podatek od drugiej krowy – tak by nie opłacało się wyprowadzać ją na pastwisko, dzięki czemu wszyscy będą wyprowadzać tylko jedną krowę i ich zysk wzrośnie do 7.
Tego rodzaju konflikty wynikają z ograniczoności wspólnych zasobów i bardzo często pojawiają się w różnych rzeczywistych sytuacjach.

6.7. Podsumowanie

Powyżej został przedstawiony wstęp do tematyki teorii gier i jej wykorzystania w systemach sztucznej inteligencji. Wykorzystanie jej do modelowania rzeczywistych problemów wiąże się z trudnościami w zdefiniowaniu wygranych, które nie zawsze mają charakter liczbowy, ani nawet mierzalny, a także często zachodzącą sprzecznością celów poszczególnych graczy. Nie bez wpływu na problemy w modelowaniu są trudności w sprecyzowaniu wiedzy poszczególnych graczy, bądź samej ich liczebności. Realizacja praktycznych systemów sztucznej inteligencji wykorzystujących teorię gier wymaga, w związku z tym, sporego wysiłku przy tworzeniu samego modelu analizowanych sytuacji. Osoby zainteresowane tą tematyką powinny przyjrzeć się przede wszystkim pozycjom literaturowym poświęconym samej teorii gier, gdyż temat wykorzystania jej w systemach sztucznej inteligencji jest bardzo rzadko poruszany w literaturze.

Bibliografia

[1] R. Bartak. On-line guide to prolog programming. http://kti.mff.cuni.cz/ bartak/prolog/. internet.

[2] Mordechai Ben-Ari. *Logika metematyczna dla informatyków*. WNT, Warszawa, 2004.

[3] D Blanchard. Two faces of ai: Aaai'97. *Expert Systems*, 15(2):120–122, 1997.

[4] R.J. Brachman and H.J. Lavesque. *Knowledge Representation and Reasoning*. Morgan Kaufmann Publishers, San Francisco, 2004.

[5] B.G. Buchanan and E.H. Shortliffe, editors. *Rule Based Expert Systems: The Mycin Experiments of the Stanford Heuristic Programming Project*. Addison-Wesley, 1984.

[6] P. Cichosz. *Systemy uczące się*. WNT, 2000.

[7] C.S. Clocksin and C.S. Mellish. *PROLOG. Programowanie*. Helion, Gliwice, 2003.

[8] W Duch. Wykłady. http://www.phys.uni.torun.pl/ wyklady/ai.

[9] C. Leondes. *Expert Systems The Technology of Knowledge Management and Decision Making for the 21st Century*. Academic Press, 2002.

[10] L Leszczyński. *Zagadnienia teorii stosowania prawa*. Zakamycze.

[11] J. Liebowitz, editor. *The handbook of applied expert systems*. CRC Press, 1998.

[12] M Malawski, A Wieczorek, and H Sosnowska. *Konkurencja i kooperacja. Teoria Gier w ekonomii i naukach społecznych*. PWN, 1997.

[13] K. Michalik. *Dokumentacja do Pakietu Sztucznej Inteligencji Sphinx*. AI-Tech.

[14] J.J. Mulawka. *Systemy ekspertowe*. WNT, Warszawa, 1996.

[15] A Niederliński. *Regułowe systemy ekspertowe*. WPKJS, 2000.

[16] A Niederliński. *Regułowo-Modelowe Systemy Ekspertowe RMSE*. PKJS, Gliwice, 2006.

[17] A Niederliński. *Programowanie w logice z ograniczeniami. Łagodne wprowadzenie dla platformy ECLiPSe*. PKJS, Gliwice, 2010.

[18] N.F. Noy and D.L. McGuinness. Abstract: Ontology development 101: A guide to creating your first ontology. http://www-ksl.stanford.edu/people/dlm/papers/ontology-tutorial-noy-mcguinness-abstract.html. internet.

[19] Z. Pawlak. Rough sets. *International J. Comp Inform Science*, (11), 1982.

[20] R. Quinlan. Induction of decision trees. *Machine Learning*, 1(1), 1986.

[21] R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[22] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Englewood Cliffs, NJ, 2nd edition edition, 2003.

[23] L. Rutkowski. *Metody i techniki sztucznej inteligencji*. PWN, 2006.
[24] L. Zadeh. Fuzzy sets. *Information and Control*, 8(3):338–353, 1965.
[25] M Zaionc, J Kozik, and M Kozik. Logika i teoria mnogości. http://wazniak.mimuw.edu.pl. internet.
[26] T. Zurek. *Komputerowe wspomaganie procesu podejmowania decyzji kredytowych*. WPKJS, Gliwice, 2005.
[27] A. Łachwa. *Rozmyty świat zbiorów, liczb, relacji, faktów, reguł i decyzji*. EXIT, 2001.