Metody Sztucznej Inteligencji

Janusz Marecki

Artificial Intelligence A Modern Approach
Stuart Russell, Peter Norvig

[...Artificial Intelligence is one of the newest disciplines. It was formally initiated in 1956, when the name was coined, although at that point work had been under way for about five years. Along with modern genetics, it is regularly cited as the "field I would most like to be in" by scientists in other disciplines. A student in physics might reasonably feel that all the good ideas have already been taken by Galileo, Newton, Einstein, and the rest, and that it takes many years of study before one can contribute new ideas. Artificial Intelligence, on the other hand, still has openings for a full-time Einstein.

The study of intelligence is also one of the oldest disciplines. For over 2000 years, philosophers have tried to understand how seeing, learning, remembering, and reasoning could, or should, be done. The advent of usable computers in the early 1950s turned the learned but armchair speculation concerning these mental faculties into a real experimental and theoretical discipline. Many felt that the new "Electronic Super-Brains" had unlimited potential for intelligence. "Faster Than Einstein" was a typical headline. But as well as providing a vehicle for creating artificially intelligent entities, the computer provides a tool for testing theories of intelligence, and many theories failed to withstand the test – a case of "out of the armchair, into a fire". Artificial Intelligence has turned out to be more difficult than many at first imagined, and modern ideas are much richer, more subtle, and more interesting as a result.

Artificial Intelligence currently encompasses a huge variety of subfields, from general – purpose areas such as perception and logical reasoning, to specific tasks such as playing chess, providing mathematical theorems, writing poetry, and diagnosing diseases. Often, scientists in other fields move gradually into artificial intelligence, where they find more tools and vocabulary to systematize and automate the intellectual tasks on which they been working all their lives. Similarly, workers in Artificial Intelligence san choose to apply their methods to any area of human intellectual endeavor. In this sense, it is truly a universal field...]

SPIS TREŚCI

Spis ilustracji

WSTĘP

Czym jest „Sztuczna Inteligencja”?

Skrypt, który czytasz drogi czytelniku jest zbiorem kilku ważniejszych zagadnień z dziedziny sztucznej inteligencji. Znajdziesz w nim wiele ciekawych zagadnień opisujących zachowania sztucznych agentów w naturalnym otoczeniu. Każde zagadnienie jest wnikliwie wytłumaczone, przez co szkice algorytmów rozwiązujących poszczególne problemy nie są trudne do zrozumienia. Na końcu każdego rozdziału znajduje się kilka zadań, których samodzielne rozwiązanie powinno ułatwić każdemu utrwalenie zdobytego materiału.

Sam temat Sztucznej Inteligencji jest tak wielki, że skrypt ten obejmuje zaledwie jego cząstkę. Generalnie materiał można podzielić na 4 grupy: Logika zdań, Logika Pierwszego Stopnia, Planowanie i Nauczanie. Są to ważne elementy wykorzystywane do budowy inteligentnie zachowujących się systemów. W skrypcie nie został poruszony ważny temat sieci neuronowych, który jest naturalnie identyfikowany ze Sztuczną Inteligencją. Zagadnienie to jest bowiem zbyt obszerne, do jego omówienia na stronach tego skryptu.

Dla kogo jest ten skrypt?

Skrypt kieruję docelowo do studentów informatyki wyższych lat, gdyż zamieszczone w nim algorytmy są bardzo abstrakcyjne i do ich poprawnego zaimplementowania czytelnik musi znać dosyć dobrze przynajmniej jeden język programowania. Samodzielne zaprogramowanie omawianych sztucznych agentów i rozwiązanie zamieszczonych w podręczniku zadań pozwala na poznanie aktualnie panujących podstaw w dziedzinie Sztucznej Inteligencji

Jak czytać ten skrypt?

By zrozumieć omawiane w skrypcie problemy nie ma potrzeby korzystania z innej literatury na ten temat. Niemniej jednak proponuję zapoznać się wszystkim z pierwszymi 2 rozdziałami, w których znajdują się terminy używane w kolejnych rozdziałach. Rozdziały 3, 4, 5 stanowią całość i powinny być czytane razem. Rozdziały 6 i 7 można czytać osobno od całości, gdyż omawiają one zagadnienia Planowania i Nauczania od podstaw.

1. Logicznie myślący agent

Ogólnie rzecz biorąc, agent logicznie myślący jest systemem wnioskującym, który pod wpływem impulsów pochodzących z zewnątrz podejmuje logiczne, ściśle określone decyzje. Każdy agent składa się z dwóch podstawowych części:

- **Baza Wiedzy** (Knowledge Base = KB), która zawiera informacje o świecie, które agent posiada podczas swojego funkcjonowania.
- **System Wnioskujący**, którego reguły korzystają z informacji zawartych w bazie wiedzy, którą mogą uaktualniać.

1.1 Agent oparty na bazie wiedzy

Jak sama nazwa wskazuje, rozpatrywany przez nas agent będzie ściśle współpracował z bazą wiedzy. Bazę wiedzy możemy rozumieć jako zbiór faktów o świecie w którym funkcjonuje agent. Znajdują się w niej **wyrażenia**, które są konkretnymi informacjami o świecie. Wyrażenie zapisane są w bazie wiedzy przy pomocy **języka reprezentacji wiedzy**, który dla pojedynczego agenta musi być jednoznaczny, jednak dla różnych agentów może odbiegać od standardu, który zostanie w tym rozdziale przedstawiony.

Posiadając bazę wiedzy musimy wyodrębnić 2 podstawowe funkcje na niej działające:

- TELL, czyli funkcja, która dodaje (lub modyfikuje) fakt do bazy wiedzy.
- ASK, czyli funkcja pytająca bazę wiedzy o daną informację, która zostaje zwrócona.

Należy zwrócić uwagę na to, że w zależności od sposobu implementacji bazy wiedzy, musimy wiedzieć jak sformułować parametry do wyżej wymienionych funkcji. Dysponując pojedynczym faktem o świecie, musimy przekształcić go do postaci „rozumianej” przez bazę wiedzy, czyli zapisać fakt przy pomocy języka reprezentacji wiedzy. Dopiero tak zbudowane wyrażenie może być przekazane jako parametr funkcji TELL.

Podobna sytuacja zachodzi dla funkcji ASK. Wyrażenie, które zostanie zwrócone przez tą funkcję nie zawsze będzie zapisane w sposób czytelny dla systemu wnioskującego. Agent będzie więc posiadał dodatkowe mechanizmy pozwalające na kodowanie i dekodowanie informacji o świecie zapisanych w bazie wiedzy.

Na poniższym schemacie przedstawione jest działanie agenta korzystającego z bazy wiedzy

```
1    function KB-AGENT(spostrzeżenie) returns akcja
2        static: KB – baza wiedzy
3                t – licznik czasu początkowo równy 0
4        TELL(KB, Stwórz-Wyrażenie(spostrzeżenie,t))
5        akcja ← ASK(KB, Stwórz-Zapytanie(t))
6        TELL(KB, Stwórz-Wyrażenie(akcja,t))
7        t ← t + 1
8        return akcja
```

Spostrzeżenie jest parametrem, który dostarcza agentowi otoczenie. Każda czynność agenta, oznaczona jako *akcja* jest więc wynikiem działania funkcji KB-AGENT. W linii 3 algorytmu do bazy wiedzy KB dodawane jest wyrażenie utworzone na podstawie *spostrzeżenia*. Ukryta funkcja Stwórz-Wyrażenie konwertuje spostrzeżenie do języka reprezentacji wiedzy.

W linii 5 wyznaczana jest nowa *akcja* agenta. W tym celu pytamy bazę wiedzy co zrobić w czasie *t,* używając do tego funkcji ASK. By przekształcić zapytanie na język reprezentacji wiedzy używamy funkcji Stwórz-Zapytanie().

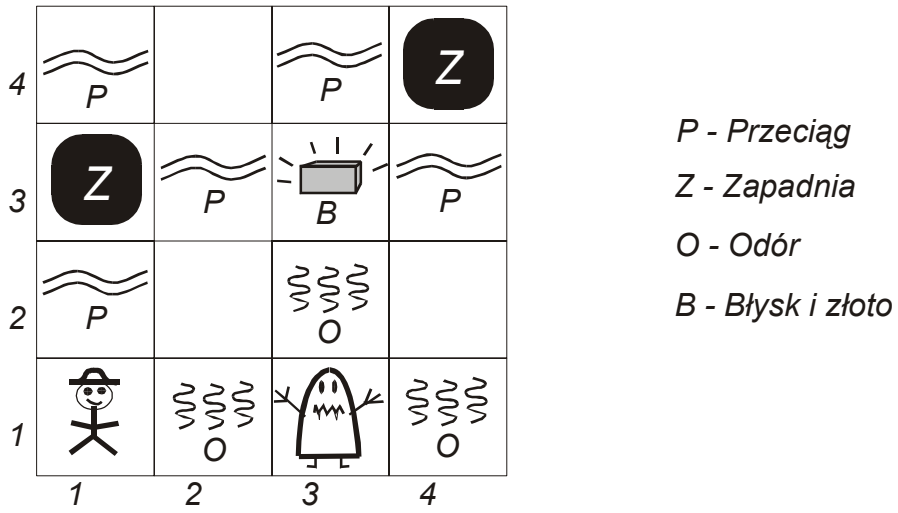
W linii 6 informujemy bazę wiedzy jaka funkcja została wykonana, a w linii 7 zwiększamy licznik czasu. Aparat wnioskujący jest ukryty w funkcjach TELL i ASK, a jego działanie wyjaśnimy później.

Baza wiedzy może posiadać pewne fakty jeszcze przez wprowadzeniem w życie agenta. Są to tzw. **deklaracje**. Reszta zawartych w niej wyrażeń jest rezultatem działania aparatu wnioskującego i spostrzeżeń w kolejnych etapach działania agenta.

1.2 Świat WUMPUS'a

Zanim przedstawimy konkretne metody logicznego wnioskowana agentów, pokażemy prosty świat wumpusa, który dostarczy nam potrzebnych skojarzeń do sposobów prezentacji wiedzy i aparatu wnioskowania.

Wumpus był wczesną grą komputerową, w której gracz wcielał się w rolę agenta, który znalazłszy się w labiryncie musiał znaleźć ukryte złoto i unikając wszelkich niebezpieczeństw wyjść z labiryntu. Labirynt składał się w komnat połączonych przejściami, po których krążyła groźna bestia. Spotkanie z nią kończyło się dla agenta śmiercią. Dodatkowo w niektórych komnatach znajdowały się zapadnie , również śmiertelne dla agenta, który odwiedził taką komnatę. W komnatach sąsiadujących z bestią agent odczuwa zapach (ang. Stench), natomiast w komnatach sąsiadujących z zapadniami agent odczuwa wietrzyk (ang. Breeze). Jako labirynt przyjmujemy siatkę, co przedstawia rysunek nr 1.



Rysunek 1 Przykładowy świat WUMPUS'a

Poniżej przedstawione są reguły rządzące omawianym światem:

- W kwadracie, w którym znajduje się złoto agent spostrzeże błysk.
- Gdy agent będzie chciał wejść na ścianę (kwadrat poza labiryntem), wówczas wyczuję blokadę.
- Agent dysponuje jedną strzałą, którą może zabić bestie.
- Gdy bestia ginie, wydaje krzyk, który daje się słyszeć w całym labiryncie
- W każdym momencie na spostrzeżenie agenta będzie się składało 5 elementów: jeśli jest odór i przeciąg natomiast nie ma błysku, blokady i krzyku, wówczas spostrzeżenie będzie miało postać: [Odór, Przeciąg, Nic, Nic, Nic]. Agent nie zna swojego położenia.
- Agent jest zawsze skierowany w jakąś stronę; może: obrócić się o 90° w lewo lub w prawo, iść wprzód, podnieść obiekt, strzelić lub wspiąć się (by wyjść z labiryntu, pod warunkiem że agent znajduje się w punkcie startowym). Wystrzelona strzała albo zabija wumpusa albo kontynuuje swój lot aż do ściany.
- Agent ginie, gdy wejdzie na pole z żywym wumpusem, bądź na pole zawierające zapadnię.
- Celem agenta jest odnalezienie złota i wyjście z labiryntu, za które otrzymuje 1000 punktów. Każdy ruch agenta kosztuje go 1 punkt, gdy agent ginie, traci 10 000 punktów.

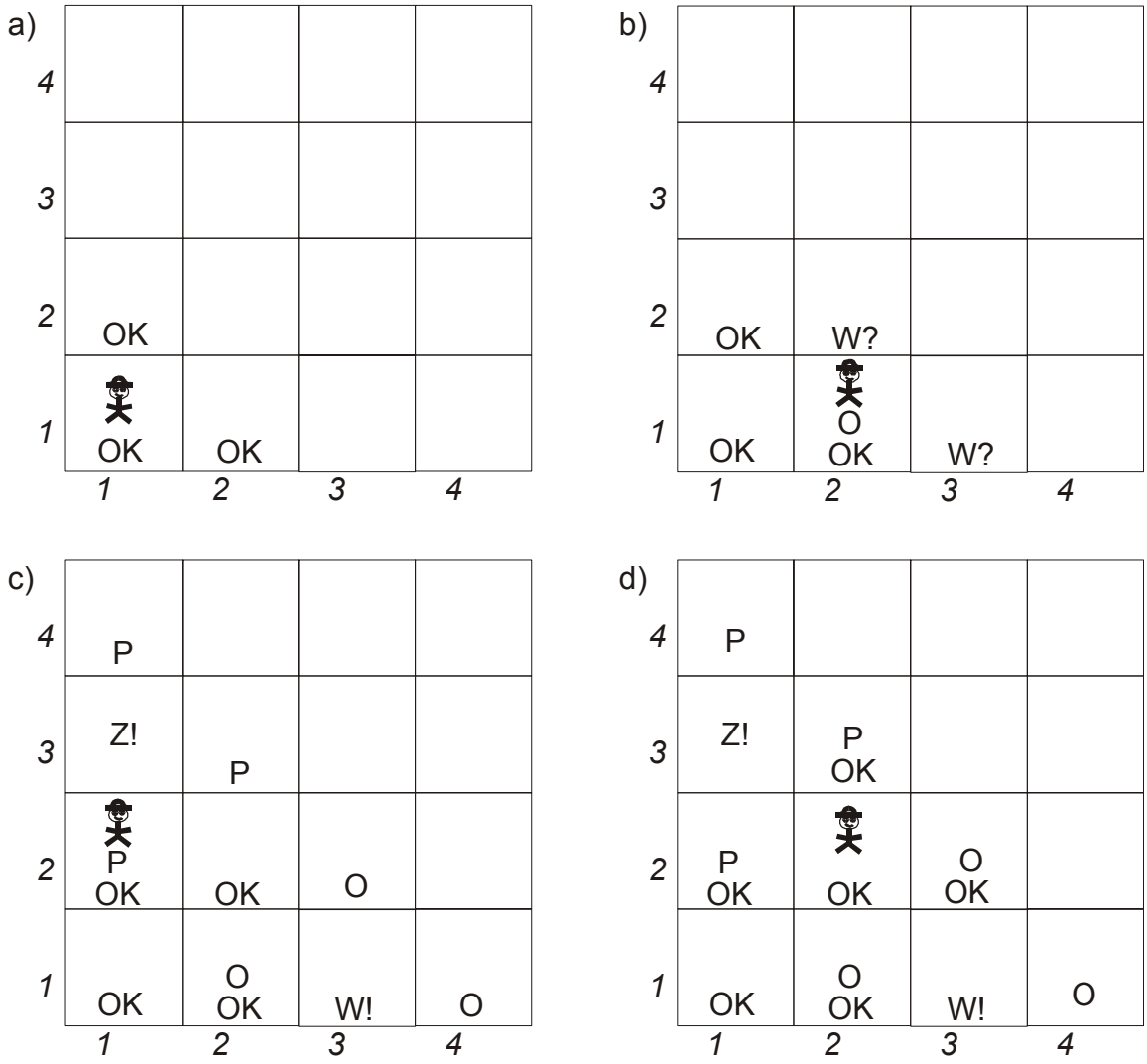
W naszym przykładzie agent startuje w kwadracie (1,1) i patrzy „do góry”, a labirynt jest kratą 4 x 4. Lokalizacje złota, wumpusa i pułapek są losowe. Agent musi w trakcie poszukiwać podejmować decyzję, czy kontynuować poszukiwania, czy też skierować się do wyjścia z pustymi rękoma. W niektórych labiryntach zdarza się, że nie ma bezpiecznej drogi do złota.

Agent nie posiada na początku żadnej wiedzy o labiryncie, zyskuje ją dopiero eksplorując kolejne lokalizacje i zapamiętując wszystko co widział w kolejnych etapach poszukiwań.

Wnioskowanie w świecie wumpusa

Agent by przeżyć musi wykazać się inteligencją, tzn. musi potrafić na podstawie jednych faktów wnioskować istnienie innych.

Przedstawimy teraz poprawne wnioskowanie agenta, który znalazł się w labiryncie przedstawionym na rysunku nr 1 w kwadracie (1,1). Na rysunku nr 2 przedstawione są kolejne etapy działania agenta:



Rysunek 2 Kolejne etapy działania agenta

Przyjmiemy następujące oznaczenia: każda kratka, o której będzie wiadomo, że jest bezpieczna będzie oznaczana jako OK. Jeśli agent coś spostrzeże (np. przeciąg), wówczas w danym kwadracie zostanie to zaznaczone. Jeśli będzie istniało prawdopodobieństwo, że w jakiejś kratce będzie wumpus, wówczas będzie ona oznaczona W? (dla zapadni Z?). Jeśli agent będzie miał pewność znalezienia wumpusa w danej kratce, to będzie w niej wstawiony znak W! (dla zapadni Z!).

Agent startuje w polu (1,1) (rysunek nr 2a) gdzie nie wyczuwa niebezpieczeństwa, daje mu to pewność, że ani w (1,2) ani w (2,1) nie ma pułapek i wumpusów; można więc te pola oznaczyć jako OK. Następnie agent odwraca się w prawo i idzie wprzód do pola (1,2) gdzie wyczuwa odór wumpusa. Agent wnioskuje więc, że w polach (1,3) lub (2,2) może znajdować się wumpus. Dalsza droga w prawo lub do góry jest niebezpieczna, więc agent odwraca się w lewo (rysunek nr 2b), wraca do pola (1,1), a następnie wchodzi do
kratki (2,1).

W kratce (2,1) jest przeciąg, czyli istnieje szansa znalezienia zapadni w polach: (2,2) i (3,1). Teraz agent wykorzystuje swój aparat wnioskowania:

Skoro będąc w kratce (1,2) agent nie wyczuł przeciągu, więc nie ma go w kratce (2,2). Jeśli w (2,1) czuć przeciąg, a nie ma go w (2,2) oznacza to, że

- W kratce (3,1) jest zapadnia
- Kratka (2,2) jest bezpieczna
- W kratkach (4,1) i (3,2) jest przeciąg.
- Wumpusa nie ma w kratce (2,2), więc jest w kratce (1,3), a w kratkach (1,4) i (2,3) jest jego odór

, co przedstawia rysunek nr 2 c.

Agent przesuwa się więc w prawo i w polu (2,2) nie wyczuwa ani przeciągu ani odoru wumpusa, stąd wnioskuje, że kratki (2,3) i (3,2) są dla niego bezpieczne co przedstawia rysunek nr 2 d).

Agent może więc kontynuować eksplorację labiryntu. Niezależnie od tego, czy znajdzie złoto, czy nie, posiada już pewną wiedzę o labiryncie, a wiedząc które pola są bezpieczne może wrócić do wyjścia.

W następnych paragrafach przedstawimy w jaki sposób budować agentów potrafiących gromadzić takie informacje jak: „Wumpus jest w polu (1,3)” lub „nie ma przeciągu w polu (2,2)” i mogących wyciągać z tych informacji logiczne wnioski.

2. Logika zdań

W rozdziale tym przedstawimy logikę zdań (logikę prostą), która pomimo swojego ograniczonego zastosowania, dobrze nadaje się do ilustracji zagadnień związanych ze sztucznymi agentami.

2.1 Składnia

Każde zdanie w logice może dawać wartość *Prawda* lub *Fałsz*. Wprowadzimy oznaczenie: *symbole zdaniowe* jako pojedyncze części zdań logicznych, które mogą przyjmować wartości prawda lub fałsz i których nie będzie dało się rozbić np.: dla zdania „Andrzej pójdzie do kina i ściana jest czerwona" mamy dwa symbole zdaniowe: P = „Andrzej pójdzie do kina" oraz Q = „Ściana jest czerwona", które mogą być prawdziwe lub fałszywe.

W logice zdań, zdania będą się składały z wartości Prawda / Fałsz, symboli zdaniowych, łączników logicznych $\neg$ oraz nawiasów ( ). Zdania te będą podlegały regułom:

- Stałe logiczne Prawda / Fałsz są same zdaniami.
- Symbole zdaniowe P, Q są same zdaniami.
- Wzięcie zdania w nawias generuje nowe zdanie np. $(P \wedge Q)$
- Zdanie (Zdanie złożone) może być utworzone przez łączenie ze sobą mniejszych zdań (Zdań złożonych lub symboli zdaniowych) przy pomocy 5 spójników logicznych: $\wedge$ koniunkcji, $\vee$ alternatywy, $\Leftrightarrow$ równoważności, $\Rightarrow$ implikacji, $\neg$negacji.

Notacja Backusa-Naura gramatyki dla logiki zdań wygląda następująco:

Zdanie	$\rightarrow$	SymbolZdaniowy | ZdanieZłożone
SymbolZdaniowy	$\rightarrow$	Prawda | Fałsz | P | Q | ... | R | ...
ZdanieZłożone	$\rightarrow$	(Zdanie) | Zdanie Spójnik Zdanie | $\neg$Zdanie
Spójnik	$\rightarrow$	$\wedge \mid \vee \mid \Leftrightarrow \mid \Rightarrow$

Przedstawiona gramatyka nie jest jednoznaczna, gdyż np. $P \vee Q$ jest równoznaczne $Q \vee P$. Priorytet wykonywania operacji jest następujący (od najwyższego do najmniejszego): Nawiasy, $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$, dlatego zdanie jest równoznaczne zdaniu : $((Q \vee P) \Rightarrow (\neg P)) \Leftrightarrow Q$.

2.2 Semantyka

Semantyka dla logiki zdań jest oczywista. Chodzi bowiem o wyznaczenie wartości zdania logicznego. Wystarczy zatem wyznaczyć wartości Symboli Zdaniowych a następnie wyznaczać wartości zdań złożonych (z uwzględnieniem priorytetów) na podstawie funkcji logicznych $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$.

Dla przykładu jeśli Q jest symbolem zdaniowym prawdziwym, a P jest fałszywym, wówczas wartość zdania $((Q \vee P) \Rightarrow (\neg P)) \Leftrightarrow Q$ obliczymy w następujący sposób: $(Q \vee P)$ jest prawdziwe, $(\neg P)$ jest prawdziwe, więc implikacja $(Q \vee P) \Rightarrow (\neg P)$ jest prawdziwa. Z tego wynika że po obu stronach równoważności mamy zdania prawdziwe i ostatecznie $((Q \vee P) \Rightarrow (\neg P)) \Leftrightarrow Q$ jest prawdziwe.

2.3 Poprawność i wnioskowanie

Wiemy już jak na podstawie danego zdania obliczyć jego wartość logiczną. Wystarczy zatem, że będziemy znali wartość logiczną symboli zdaniowych, a z łatwością obliczymy wartość logiczną danej funkcji zdaniowej.

Możemy powyższe rozumowanie odwrócić. Załóżmy, że mamy daną funkcję zdaniową, która jest poprawna, np.:

$((P \vee Q) \wedge \neg Q) \Rightarrow P$

Widzimy, że niezależnie od wartości logicznych P i Q, całe wyrażenie jest poprawne. P może oznaczać, że wumpus jest w kwadracie (1,3), a Q, że jest w kwadracie (2,2). Jeśli w pewnym momencie agent dowie się, że zachodzi $P \vee Q$, a później, że zachodzi $\neg Q$, wówczas możemy wydedukować, że prawdą jest również P.

Wynika stąd, że możemy znaleźć wumpusa na podstawie innych informacji, o których potrafimy stwierdzić czy są prawdziwe, czy fałszywe.

Załóżmy teraz, że mamy n symboli zdaniowych: P_1, P_2, ..., P_n. Łatwo zauważyć, że istnieje 2^n wartościowań dla tych symbolów (różnych sposobów przyporządkowania kolejnym symbolom wartości Prawdy lub Fałszu.

Wyobraźmy sobie teraz różne funkcje logiczne n- parametrowe, których dziedziną jest P_1, P_2, ..., P_n. Dwie takie funkcje f_i i f_j są różne, jeśli istnieje przynajmniej jedno wartościowanie, dla którego wartości tych funkcji są różne. Dla 2^n wartościowań mamy zatem 2^{2^n} różnych funkcji logicznych. Należy zauważyć, że funkcje nie muszą zależeć od wszystkich parametrów, np. $P_1 \wedge P_2$ lub $\neg P_2$ są funkcjami logicznymi. Na rysunku nr 3 przedstawiona jest tabela wartościowań i funkcji n symboli zdaniowych: P_1, P_2, ..., P_n. Dla uproszczenia Prawdę będziemy oznaczali jako 1, a Fałsz jako 0.

	Symbole zdaniowe					Funkcje logiczne				
	P_1	P_2	...	P_{n-1}	P_n	f_1	f_2	f_3	…	f_m
2^n Wartościowań	0	0	…	0	0	0	1	0	…	1
	0	0	…	0	1	0	0	1	…	1
	0	0	…	1	0	0	0	0	…	1
	0	0	…	1	1	0	0	0	…	1
	…	…	…	…	…	…	…	…	…	…
	1	1	…	1	0	0	0	0	...	1
	1	1	…	1	1	0	0	0	…	1
						2^{2^n} funkcji				

Rysunek 3 Tabela wartościowań i funkcji

Rozpatrzmy przypadek funkcji $((P \vee Q) \wedge \neg Q) \Rightarrow P$,którym zajmowaliśmy się wcześniej. Oznaczmy tą funkcję jako f_i, P jako P_u, Q jako P_v. Przyjmijmy także $f_j = (P \vee Q)$ oraz $f_k = \neg Q$. Jeśli agent dowiedział się, że $P \vee Q$ jest prawdziwe, to znaczy że istnieje takie wartościowanie P_1 .. P_n ,dla którego f_j jest prawdziwe. Oznacza to, że w tabeli na rysunku nr 3 są takie wiersze, dla których f_j przyjmuje wartość 1. Niech zbiór tych wierszy będzie Z_1. Wiersze te reprezentują wartości, które mogą przyjmować symbole zdaniowe. Jeśli rozpatrzymy teraz funkcję f_k ,to znów sprawdzamy czy istnieją wiersze (wartościowania) dla których wartość f_k wynosi 1. Zbiór tych wierszy oznaczmy jako Z_2 . Teraz wiersze, które będą zawierały dopuszczalne wartościowania będą się znajdowały w przecięciu zbiorów $Z_1 \cap Z_2$. Jeśli okaże się, że dla wszystkich wierszy $Z_1 \cap Z_2$ jakiś symbol zdaniowy będzie zawsze taki sam, oznacza to, że znaleziona została jego ostateczna wartość logiczna.

Dla rozpatrywanej funkcji f_i zbiór $Z_1 \cap Z_2$ zawierał zatem wiersze, dla których symbol P_u był zawsze taki sam, czyli wywnioskowane zostało P.

Algorytm polega więc na analizowaniu poszczególnych funkcji logicznych i wyszukiwaniu wierszy, dla których te funkcje przyjmują wartość 1. Jeśli wśród tych wierszy jakiś symbol jest już ustalony, to znamy jego ostateczną wartość logiczną. Jeśli zbiór wierszy jest pusty, to wystąpił błąd wartości jakiejś funkcji spośród f_1 .. f_m, np. prawdą była funkcja $P \wedge Q$ oraz

funkcja $\neg Q$, co jest niedopuszczalne. Oznacza to, że agent posługuje się sprzecznymi metodami wnioskowania, np. na podstawie wiadomości, że jest odór w kwadracie (1,2) wnioskuje jednocześnie, że Wumpus jest w kwadracie (1,3) ,oraz że wumpusa nie ma w kwadracie (1,3).

Mimo iż taki algorytm wnioskowania nie jest trudny do zaprogramowania, to jednak prawie nigdy się go nie stosuje, ponieważ dla 2^n wartościowań jego złożoność obliczeniowa jest ekspotencjalna.

2.4 Zasady wnioskowania dla logiki zdań

Przedstawiony algorytm wnioskowania jest często rozbudowywany o inne zasady wnioskowania, które na podstawie pewnych zdań logicznych mogą wnioskować prawdziwość innych zdań. Wprowadzimy następujące oznaczenie:

α / β , które oznacza, że jeśli zdanie α może być wydedukowane z bazy wiedzy KB, to z KB wydedukowane może być także zdanie β. Składnia α, β oznacza, że wywnioskowane zostały: zdanie α oraz zdanie β. Poniżej przedstawionych jest 7 metod wnioskowania dla logiki zdań:

- **Modus-Ponens** lub **Implikacja-Eliminacja** (Z implikacji i poprzednika implikacji wnioskujemy następnika implikacji)
  $\alpha \Rightarrow \beta,\ \alpha\ /\ \beta$
- **Eliminacja koniunkcji** (z koniunkcji zdań wnioskujemy poszczególne składniki koniunkcji)
  $\alpha_1 \wedge \alpha_2 \wedge ... \wedge \alpha_n\ /\ \alpha_i$
- **Rozszerzenie koniunkcji** (z listy zdań wnioskujemy nowe zdanie, które jest ich koniunkcją)
  $\alpha_1 , \alpha_2 , ... , \alpha_n\ /\ \alpha_1 \wedge \alpha_2 \wedge ... \wedge \alpha_n$
- **Rozszerzenie alternatywy** (Zdanie można rozszerzyć przez alternatywę o inne składniki)
  $\alpha_i\ /\ \alpha_1 \vee \alpha_2 \vee ... \vee \alpha_n$
- **Eliminacja podwójnej negacji** (Ze zdania podwójnie zanegowanego można wywnioskować zdanie prawdziwe)
  $\neg\neg\alpha\ /\ \beta$
- **Rozdzielenie jednostki** (Jeśli jeden z ze składników prawdziwej alternatywy jest fałszywy, to drugi jest prawdziwy)
  $\alpha \vee \beta,\ \neg\alpha\ /\ \beta$
- **Rozdzielenie** (Ponieważ β nie może być naraz prawdą i fałszem, co najmniej jeden z pozostałych składników dwóch alternatyw jest prawdziwy)
  $\alpha \vee \beta,\ \neg\beta \vee \delta\ /\ \alpha \vee \delta$ lub równoznaczne

$$\neg\alpha \Rightarrow \beta,\ \beta \Rightarrow \delta \quad / \quad \neg\alpha \Rightarrow \delta$$

Dowód logiczny jest więc wynikiem kolejnych zastosowań powyższych reguł poczynając od zdań znajdujących się w bazie wiedzy i kończąc na zdaniu, którego prawdziwość jest poszukiwana.

Jeśli wszystkie zdania w bazie wiedzy mają znaną postać, np. Klauzul Horn'a $P_1 \wedge P_2 \wedge ... \wedge P_n \Rightarrow Q$, gdzie P_i i Q są symbolami zdaniowymi, wówczas istnieje procedura wnioskująca działająca w czasie wielomianowym.

2.5 Problem wumpusa w logice zdań

Pokażemy teraz w laki sposób powinien się zachowywać agent w „świecie wumpusa" przy wykorzystaniu logiki zdań.

Baza Wiedzy

W każdym etapie działania agenta jego spostrzeżenia są przekształcane na zdania i zapisywane w bazie wiedzy. Skupimy się teraz na etapie przedstawionym na rysunku nr 2c. Celem jest by agent wywnioskował, że w kwadracie (2,2) nie ma wumpusa. Dotychczasowa wiedza agenta będzie przedstawiona w postaci skrótowej tzn. zdanie: „W kratce (1,2) jest odór" będzie oznaczone $O_{1,2}$. Na początku etapu w bazie wiedzy agenta będą więc następujące spostrzeżenia:

$O_{1,2}$
$\neg O_{1,1}$
$\neg O_{2,1}$

$\neg P_{1,2}$
$\neg P_{1,1}$
$P_{2,1}$

W dodatku, agent musi dysponować podstawową wiedzą dotyczącą otoczenia (w tym przypadku musi znać reguły świata wumpusa przedstawione na początku rozdziału 1). Wiemy, że jeśli w danej kratce nie ma odoru, to w żadnej kratce sąsiadującej nie ma wumpusa. Dla logiki zdań fakt ten moglibyśmy przedstawić rozpatrując kolejno wszystkie kratki labiryntu, co jest jednak pracochłonne. Dla wydedukowania następnego posunięcia agenta potrzebne nam będą tylko reguły dotyczące kratek, o których coś już wiemy:

Reguła 1: $\neg O_{1,1} \Rightarrow \neg W_{1,1} \wedge \neg W_{1,2} \wedge \neg W_{2,1}$

Reguła 2: $\neg O_{1,2} \Rightarrow \neg W_{1,1} \wedge \neg W_{1,2} \wedge \neg W_{1,3} \wedge \neg W_{2,2}$

Reguła 3: $\neg O_{2,1} \Rightarrow \neg W_{1,1} \wedge \neg W_{2,1} \wedge \neg W_{3,1} \wedge \neg W_{2,2}$

Będziemy potrzebowali także zdania, które da nam możliwe położenia wumpusa:

Reguła 4: $O_{1,2} \Rightarrow W_{1,1} \vee W_{1,2} \vee W_{1,3} \vee W_{2,2}$

Znajdowanie wumpusa

Do znalezienia wumpusa wykorzystujemy algorytm przedstawiony w rozdziale 2.3. Pamiętajmy, że musimy uwzględnić wszystkie dotychczasowe symbole zdaniowe, czyli $W_{1,1}$, $W_{1,2}$, $W_{1,3}$, $W_{2,2}$, $O_{1,1}$, $O_{1,2}$, $O_{2,1}$, $P_{1,1}$, $P_{1,2}$, $P_{2,1}$. Dla wszystkich możliwych wartościowań tabela przedstawiona na rysunku nr 3 posiadałaby 2^{12} = 4096 wierszy, skupimy się jednak tylko na tych wierszach, które pozwolą nam na znalezienie wumpusa w kwadracie (1,3) , chcemy więc, by algorytm znalazł wiersz, w którym wartość logiczna $W_{1,3}$ jest 1.

Na początku wyeliminujemy kratki, w których wumpusa nie może być na podstawie znanych spostrzeżeń. Wykorzystując zasadę Modus Ponens dla $\neg O_{1,1}$ w regule nr 1 otrzymujemy zdanie:

$\neg W_{1,1} \wedge \neg W_{1,2} \wedge \neg W_{2,1}$

, które pod wpływem Eliminacji koniunkcji daje nam zdania:

$\neg W_{1,1}$ $\neg W_{1,2}$ $\neg W_{2,1}$

Wykorzystując zasadę Modus Ponens dla $\neg O_{2,1}$ w regule nr 3 ,a następnie Eliminację koniunkcji otrzymujemy dodatkowo:

$\neg W_{2,1}$ $\neg W_{3,1}$ $\neg W_{2,2}$

Zasada Modus Ponens dla $O_{1,2}$ w regule nr 4 daje:

$W_{1,1} \vee W_{1,2} \vee W_{1,3} \vee W_{2,2}$

Stosujące teraz trzykrotnie zasadę rozdzielenia jednostki otrzymujemy kolejno:

$W_{1,2} \vee W_{1,3} \vee W_{2,2}$

$W_{1,2} \vee W_{1,3}$ i ostatecznie pożądany fakt:

$W_{1,3}$

Przekształcenie wiedzy w akcję:

Nasz agent w świecie wumpusa kierował się zasadami „zdrowego rozsądku” tzn. odwiedzał tylko kwadraty, które były bezpieczne. Istnieją różne strategie działania agenta (np. ryzykowna – dopuszczająca odwiedzanie niepewnych kwadratów) a ogólną procedurę działania można zapisać w następujący sposób:

```
   function AGENT-LOGIKA-ZDAN(spostrzeżenie) returns akcja
1        static: KB – baza wiedzy
2                t – licznik czasu początkowo równy 0
3        TELL(KB, Stwórz-Wyrażenie(spostrzeżenie,t))

4        for each akcja in lista-możliwych-akcji do
5                if ASK(KB, Stwórz-Zapytanie(t,akcja)) then
6                        t ← t + 1
7                        return akcja
8        end
```

Pętla w linii 4 pozwala nam tylko na znalezienie pierwszej akcji, która jest dopuszczalna. Za sprawdzenie, czy akcja jest dopuszczalna odpowiedzialna jest procedura ASK z linii nr 4. W linii 7 zwracana jest pierwsza dopuszczalna akcja.

By zdefiniować reguły dla agenta chcącego się poruszać musimy jeszcze dodać 4 symbole zdaniowe odpowiedzialne za jego orientację: $A_{Góra}$, $A_{Dół}$, A_{Lewo}, A_{Prawo} oraz symbol zdaniowy *Wprzód*. Teraz możemy napisać już konkretną regułę:

$A_{2,1} \wedge A_{góra} \wedge Z_{3,1} \Rightarrow \neg$Wprzód

Widzimy więc, że takie reguły pozwalają agentowi odpowiedzieć na takie pytania jak: „Czy mogę iść do przodu?” , a nie na pytania jak typu „Jaką akcję wykonać? ”.

Logika zdań znajduje bardzo wąskie zastosowanie, ponieważ nawet dla światów małych rozmiarów, dla których ilość symboli zdaniowych jest n, ilość wierszy w tabeli wartościować (rysunek nr 3) jest 2^n, a co za tym idzie wnioskowanie jest wykonywane w czasie ekspotencjalnym.

Kolejnym problemem jest zagadnienie zmiany świata. Jeśli np. wumpus się porusza, wtedy chcielibyśmy, by agent w różnych chwilach reagował inaczej. Podobnie, nawet dla statycznego świata może się okazać, że agent będzie wykonywał czynności cykliczne i jego działanie się zapętli. By uniknąć tych sytuacji, każdy symbol zdaniowy będzie musiał był inny dla kolejnych czasów. To stwarza dwie niedogodności: po pierwsze, nie wiemy jak długo gra będzie trwała, a po drugie, musimy przepisać wszystkie reguły w zmodyfikowanych wersjach, np.:

$A_{2,1}^{1} \wedge A_{\text{góra}}^{1} \wedge Z_{3,1}^{1} \Rightarrow \neg \text{Wprzód}^{1}$
$A_{2,1}^{2} \wedge A_{\text{góra}}^{2} \wedge Z_{3,1}^{2} \Rightarrow \neg \text{Wprzód}^{2}$
$A_{2,1}^{3} \wedge A_{\text{góra}}^{3} \wedge Z_{3,1}^{3} \Rightarrow \neg \text{Wprzód}^{3}$
.....

Oznacza to, że dla gry trwającej 100 etapów, będziemy mieli 100 reguł mówiących, że jeśli agent jest w kratce (2,1) i patrzy do góry, a tam jest zapadnia, to nie powinien posuwać się do przodu.

W następnym rozdziale przedstawimy logikę pierwszego stopnia, która pozwala zredukować tych 100 reguł, do jednej.

2.6 Zadania

1. W rozdziale powiedzieliśmy, że aby wyrazić regułę: „Nie idź do przodu, jeśli jest przed tobą wumpus" potrzeba 64 zdań logicznych. Co by się stało, gdybyśmy ten fakt wyrazili pojedynczą zasadą:
WumpusPrzedAgentem $\Rightarrow \neg$*DoPrzodu*

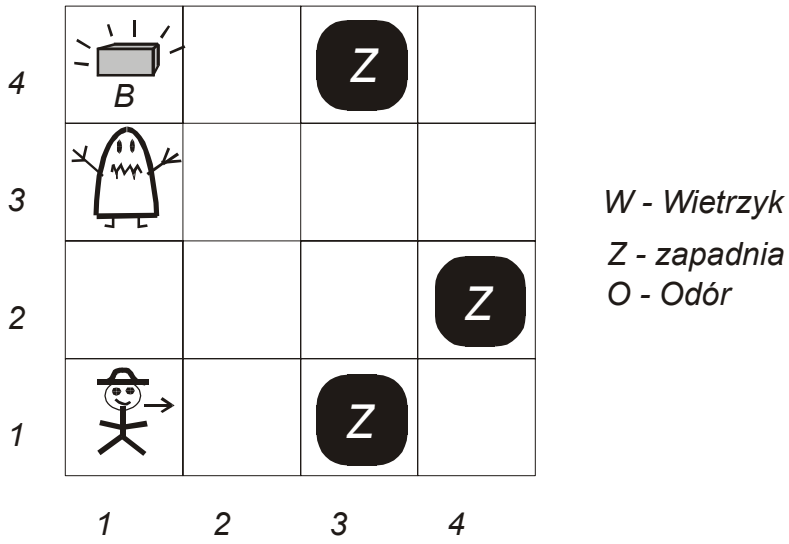
Czy jest to wykonalne? Jaki byłby wpływ tego zdania na resztę bazy wiedzy?

2. Jan, Szymon i Cezar okupują w firmie stanowiska: programista, akwizytor i menedżer (niekoniecznie w tej kolejności). Jan jest winny programiście 100zł. Małżonka menedżera zabrania, by pożyczał pieniądze. Szymon nie jest żonaty. *Pytanie: kto posiada jakie stanowisko?*

Przedstaw fakty w logice zdań. Powinno być 9 symboli zdaniowych dla przypisania stanowisk wszystkim osobom (przykład JA oznacza, że Jan jest akwizytorem). Dla wyrażenia faktu , że np. Jan pożyczania pieniądze i że jest żonaty można użyć zdania J_P $\wedge$ J_Ż itd. Koniunkcja początkowych faktów tworzy bazę wiedzy. Jest 6 możliwości skojarzeń osoby/stanowiska. Znajdź właściwe skojarzenie.

3. Zbuduj przy pomocy Notacji Backusa-Naura gramatykę reprezentującą wszystkie adresy internetowe protokołu HTTP.

4. Agent znalazł się w świecie wumpusa przedstawionym Na rysunku nr 4. Zakładamy, że na starcie nie wie niczego o tym świecie (nie zna położenia złota, puapek i wumpusa). Czy może on bez ryzyka znaleźć złoto? Odpowiedź uzasadnij i przedstaw bazę wiedzy agenta w kolejnych krokach poszukiwań.



Rysunek 4 Zadanie znalezienia złota

3. Logika pierwszego stopnia

W tym rozdziale zajmiemy się zagadnieniem Logiki pierwszego stopnia, która jest silniejsza i bardziej efektywna od logiki zdań. Główną jej cechą jest to, że świat składa się z obiektów, które posiadają identyfikujące indywidualne własności, które odróżniają je od innych obiektów.

Wśród tych obiektów istnieją relacje, wśród których wyróżniamy klasę funkcji, które dla danego argumentu dają jedną wartość.

Okazuje się, że prawie każdy świat da się przedstawić przy pomocy takiego rozróżnienia. Poniżej widać przykładowe obiekty, ich właściwości i relacje:

- Obiekty: drzewa, gry, dni tygodnia, ludzie, kolory ...
- Relacje: Większy od, posiada, Brat od, wewnątrz ...
- Własności: niebieski, prostokątny, siódmy ...
- Funkcje: Ojciec od, najwyższy, głębokość ...

Poniższe przykłady pokazują, że prawie każde zdanie może być przedstawione przy pomocy logiki pierwszego stopnia:

- „Pięć razy pięć równa się dwadzieścia pięć" – Obiekty: *pięć, pięć razy pięć, dwadzieścia pięć*; Relacje: *równa się*; Funkcje*: plus. Obiekt pięć razy pięć jest rezultatem działania funkcji* razy na obiektach *pięć i pięć*; *dwadzieścia pięć jest* inną nazwą tego obiektu
- „W kwadratach sąsiadujących z zapadnią jest przeciąg" . Obiekty: *zapadnia*, *kwadrat*; własności: *posiada przeciąg*; Relacje: *sąsiaduje z*.
- „W roku 1968 Miss Świata była piękna Jenny Doll" Obiekty: *rok 1968, Miss Świata, Jenny Doll*; Relacja: *być* ; Własności: *piękna*.

W logice pierwszego stopnia nie są faworyzywane żadne kategorie obiektów i nie jest sprawdzana przynależność funkcji do konkretnych obiektów, co nie zmniejsza jej popularności i daje programiście duże pole do działania. Zastosowanie np. funkcji mnożenia do parametrów Zdzisław i Zbigniew mogłoby dać nam inny obiekt bez naruszenia zasad logiki pierwszego stopnia.

3.1 Składnia i Semantyka

W logice zdań każde wyrażenie logiczne było zdaniem. W logice pierwszego stopnia spotykamy się z termami, które zbudowane są z symboli stałych, ze zmiennych oraz z symboli funkcyjnych. Do budowy zdań używane

są kwantyfikatory i predykaty działające na termach. Gramatyka logiki pierwszego stopnia w notacji Backusa Naura wygląda następująco:

Zdanie	→	ZdanieAtomowe | Zdanie łącznik Zdanie | Kwantyfikator Zmienna, ... Zdanie | ¬Zdanie | (Zdanie)
ZdanieAtomowe	→	Predykat(Term, ...) | Term = Term
Term	→	Funkcja(Term, …) | Stała | Zmienna
Spójnik	→	$\wedge \mid \vee \mid \Leftrightarrow \mid \Rightarrow$
Kwantyfikator	→	$\forall \mid \exists$
Stała	→	B | X_{21} | Andrzej | ...
Zmienna	→	b | z | s | ...
Predykat	→	Brat od | ma kolor | ...
Funkcja	→	Ojciec od | Głębokość | ...

Na początku wyjaśnimy pojęcie **Zdania i ZdaniaAtomowego**. Z dużym uproszczeniem moglibyśmy napisać, że Zdanie jest połączeniem spójnikami i kwantyfikatorami pewnej ilości ZdańAtomowych. W istocie jeśli przyjrzymy się pierwszej produkcji w powyższej gramatyce, to widzimy, że na początku ma miejsce wielokrotne produkowanie nieterminala „Zdanie". By przerwać generowanie nieterminala Zdanie użyta jest produkcja Zdanie → ZdanieAtomowe, a to drugie z kolei rozbija się na predykaty i termy.

Symbole stałe: Obiekty w świecie posiadają swoje odnośniki wśród symboli stałych. Każdy symbol stały nadaje nazwę dokładnie jednemu obiektowi, obiekt natomiast nie musi posiadać nazwy, lub posiadać wiele nazw. Dla przykładu weźmy symbol „Andrzej", który w jednej interpretacji może

odnosić się do boksera: Andrzej Gołota, który odnosił sukcesy pod koniec XX wieku, a w innej do Andrzeja Nowaka, nauczyciela Biologii w Bytomiu.

Predykaty: odnoszą się zawsze do konkretnej relacji na obiektach. Dla przykładu weźmy predykat „Wróg", który dla jednego obiektu A zwraca obiekty reprezentujące wrogów obiektu A. Relacja „wróg" jest implementowana jako zbiór par, przy czym każda para jest uporządkowana. Weźmy np. osoby: Adam Kowalski, Paweł Nowak i Krzysztof Flotysz. Jeśli każda osoba jest wrogiem dla innej osoby, to relacja wróg będzie wyglądała następująco:

{<Adam Kowalski, Paweł Nowak>,
<Paweł Nowak, Krzysztof Flotysz> }

Funkcje: Każda funkcja jest relacją, z tą tyko różnicą, że dla jednego argumentu funkcji istnieje jedna i tylko jedna wartość. Weźmy np. funkcję „Głębokość", której dziedziną są wszystkie baseny w Polsce. Funkcja ta dla dowolnego basenu zwróci jego średnią głębokość. Niemożliwe jest, by jeden basen posiadał 2 różne średnie głębokości, więc funkcja jest reprezentowana jako zbiór takich par, w których pierwszym elementem jest argument funkcji, a drugim wartość na tym argumencie np.:

{<Basen sportowy „Górnik" , 2.20> , <"Jaworze" , 1.80> }
Do powyższego zbioru nie można dodać pary:
<"Jaworze", 1.70>

Termy: są wyrażeniami logicznymi, które odnoszą się do obiektów. Zgodnie z tym co zostało napisane wcześniej, wszystkie symbole stałe są termami. Czasami wygodniej jest odnosić się do pewnych obiektów nie przez ich nazwy, ale przez wyrażenie logiczne, które na nie wskazują. Weźmy dla przykładu term: „Prawy but od". Jeśli chcielibyśmy wyrazić jakiś fakt o prawych butach wśród grupy ludzi, to bez termów bylibyśmy zmuszeni nadawać nazwy każdemu prawemu butowi każdej osoby z grupy. Przy użyciu termów na te buty mogą wskazywać zdania „Prawy but od Wojtka" lub „Prawy but od Marty". Możemy utożsamiać termy ze wskaźnikami do obiektów. Złożenie różnych termów jest więc złożeniem wskaźników.

Zdania złożone: Możemy do budowy zdań złożonych używać spójników, podobnie jak to miało miejsce w logice zdań. Semantyka tak utworzonych zdań jest taka sama jak w przypadku logiki zdań. Przykładowo:

- Wróg(Adam, Paweł) $\wedge$ Wróg(Paweł, Krzysztof) jest prawdziwe gdy wszystkie 3 osoby są dla siebie wzajemnie wrogami.
- Głębokość(„Jaworze", 1.80) $\Rightarrow$ $\neg$Głębokość(„Jaworze", 1.70). Czyli gdy basen ma głębokość 1.80 m., to nie ma głębokości 1.70 m.

- $\neg$(Wyższy(„Jan”, 1.85) $\vee$ Niższy(„Jan”, 1.60)) Jest prawdziwe, gdy wzrost Jana jest w przedziale [1.60 .. 1.85].

Kwantyfikatory:

Jedną z podstawowych cech logiki pierwszego stopnia, która jest zarazem ogromną zaletą w porównaniu do logiki zdań są kwantyfikatory. Pozwalają one na wyrażenie wielu faktów logicznych przy użyciu jednego zdania. W logice pierwszego stopnia używane są dwa kwantyfikatory:

Kwantyfikator uniwersalny $\forall$.

Rozpatrzmy problem znalezienia złota w świecie wumpusa. Jak wiadomo jeśli Agent znajdujący się w danej kratce spostrzeże błysk, to powinien wywnioskować, że w tej kratce znajduje się złoto. W logice zdań zasadę tą reprezentowałyby reguły:

$\text{Błysk}_{1,1} \Rightarrow \text{Złoto}_{1,1}$
$\text{Błysk}_{1,2} \Rightarrow \text{Złoto}_{1,2}$
$\text{Błysk}_{1,3} \Rightarrow \text{Złoto}_{1,3}$
.....

W logice pierwszego stopnia zasadę tą opisuje jedno zdanie, które opisuje nam zależność między obiektami, bez konieczności wymieniania ich nazw :

$\forall_{\text{kwadratu x}}$ Błysk(x) $\Rightarrow$ Złoto(x)

co oznaczą, że dla każdego kwadratu x (który będzie posiadał dwie współrzędne, których tutaj nie zapisujemy) jeśli w x jest Błysk, to w x jest złoto.

Podstawiając za x kolejne kwadraty otrzymalibyśmy n^2 zdań logicznych, gdzie n jest szerokością (i zarazem długością) labiryntu. Z naszego punktu widzenia interesujące jest zdanie, w którym coś zostanie wydedukowane tzn. niech z kwadracie (3,3) agent spostrzeże błysk. Zdanie

Błysk(3,3) $\Rightarrow$ Złoto(3,3)

Daje mu zatem pewność, że w kwadracie (3,3) jest złoto, ponieważ poprzednik implikacji jest prawdziwy i aby całe zdanie było prawdziwe, także następnik implikacji musi być prawdziwy. W pozostałych zdaniach poprzednik implikacji jest fałszywy, przez co nie możemy powiedzieć niczego o następniku implikacji.

Kwantyfikator uniwersalny $\forall_{\text{term X}}$ tworzy zatem zdanie o wszystkich obiektach wskazywanych przez X znajdujących się w świecie, ale jego rezultat nie musi mieć wpływu na wszystkie te obiekty.

Kwantyfikator Istnienia ∃

Kwantyfikatory uniwersalne tworzyły zdania o obiektach w świecie bez konieczności ich nazywania. Na podobnej zasadzie działa kwantyfikator ∃, który tworzy jedno zdanie o jakimś obiekcie bez konieczności znania jego nazwy.

Załóżmy, że chcielibyśmy zapisać, że Monika ma przyjaciela, który jest aktorem. Wykorzystujemy w tym zdaniu kwantyfikator ∃:

$\exists x$ Przyjaciel(x, Monika) $\wedge$ Aktor(x)

Ostatnie zdanie czytamy: istnieje taka osoba x, która jest przyjacielem Moniki i która jest aktorem. Jest to więc równoznaczne zdecydowanie dłuższej formule, w której za x podstawiamy wszystkie znane osoby:

(Przyjaciel(Adam, Monika) $\wedge$ Aktor(Adam))$\vee$
(Przyjaciel(Dorota, Monika) $\wedge$ Aktor(Dorota))$\vee$
(Przyjaciel(Paweł, Monika) $\wedge$ Aktor(Paweł))$\vee$
.....itp.

Zdanie z wykorzystaniem kwantyfikatora ∃ jest prawdziwe, jeśli przynajmniej jedno z powyższych składników złożonej alternatywy jest prawdziwe.

Tak jak w przypadku kwantyfikatora uniwersalnego ∀ naturalnym wydawało się używanie spójnika $\Rightarrow$, tak w przypadku kwantyfikatora ∃ naturalnym wydaje się użycie spójników $\wedge$ i $\vee$. Połączenie ∃ oraz $\Rightarrow$ daje w rezultacie słabe zdanie logiczne, prawie zawsze prawdziwe:

$\exists x$ $\text{Wyrażenie}_1(x) \Rightarrow \text{Wyrażenie}_2(x)$

Zdanie to **nie** jest spełnione tylko wtedy, gdy dla każdego x zachodzi:

$\neg\text{Wyrażenie}_2(x)$ i $\text{Wyrażenie}_1(x)$

Często dla wyrażenia, że istnieje tylko jeden obiekt o danej właściwości dodajemy po kwantyfikatorze ∃ symbol ! .Symbol ten jest użyteczny w wielu zdaniach logicznych, np. $\exists!x \; 3+x = 4$

Połączenie ∀ i ∃.

Kwantyfikatory: Uniwersalny i Istnienie są ze sobą połączone przez negację. Rozpatrzmy przykładowe zdanie: „Wszyscy pasażerowie są już w samolocie" jest ono równoważne zdaniu: „Nie ma pasażerów, którzy są poza samolotem". Jeśli więc dla pasażera x, zdanie P(x) oznacza, że jest on w samolocie, mamy równoważność:

$\forall x \; P(x) \equiv \neg\exists x \neg P(x)$

Ponieważ „∀" reprezentuje w świecie koniunkcję zdań, a „∃" alternatywę, to na podstawie praw De Morgan'a mamy:

$$\forall x\ \neg P(x) \quad \equiv \quad \neg\exists x\ P(x)$$
$$\neg\forall x\ P(x) \quad \equiv \quad \exists x\ \neg P(x)$$
$$\forall x\ P(x) \quad \equiv \quad \neg\exists x\ \neg P(x)$$
$$\exists x\ P(x) \quad \equiv \quad \neg\forall x\ \neg P(x)$$

Stąd do reprezentacji świata w logice pierwszego stopnia wystarczyłby jeden z tych kwantyfikatorów i symbol negacji.

Równość

W logice pierwszego stopnia jest jeszcze jeden sposób na tworzenie Zdań Atomowych. By wyrazić, że dwa termy odnoszą się do tego samego obiektu używamy znaku równości np.:

Brat(Ewa) = Adam

oznacza, że term Brat(Ewa) odnosi się do tego samego obiektu co term Adam.

Na równość możemy patrzyć jak na pewien specyficzny predykat składający się z takich par, w których dwa obiekty pary są takie same, czyli innymi słowy relację identyczności np.:

{ <Adam, Adam>, <1.80,1.80>, <fiolet, fiolet>, <Niemcy, Niemcy> ... }

stąd chcąc sprawdzić, czy Głębokość(„Jaworze") = 1.80 , najpierw szukany jest obiekt wskazywany przez term Głębokość(„Jaworze") , a następnie jest on porównywany z obiektem 1.80.

3.2 Logika wyższego stopnia

Logika pierwszego stopnia charakteryzuje się tym, że używane w niej kwantyfikatory odnoszą się do obiektów, a nie do relacji, czy funkcji działających na tych obiektach.

W logikach wyższego rzędu możemy używać kwantyfikatorów zarówno do obiektów, jak i do funkcji i relacji; Możemy np. stwierdzić, że dowolne obiekty są takie same, jeśli posiadają takie same cechy:

$\forall o_1, o_2 \quad (o_1=o_2) \Leftrightarrow \forall_{\text{cecha } c}\, c(o_1) = c(o_2)$

Logiki wyższych stopni są silniejsze od logiki pierwszego stopnia, jednak coraz trudniejsze staje się wnioskowanie zapisanych w nich zdań.

3.3 Używanie logiki pierwszego stopnia

Dziedzina zbiorów:

Przedstawimy teraz przykład zastosowania logiki pierwszego stopnia dla świata zbiorów. Chcemy, by nasza logika rozróżniała poszczególne zbiory (w tym zbiór pusty) ,by wykrywała, czy element należy do zbioru, by wykrywała, czy element jest zbiorem, by była możliwa budowa nowych zbiorów, dodawanie elementów do zbiorów oraz wyodrębnianie z nich podzbiorów.

Tak jak na początku tego rozdziału, należy na początku wyodrębnić:

- Zmienne: będą nimi wszystkie niepuste zbiory w świecie
- Stałe : będzie nim *ZbiórPusty*.
- Predykaty: Element(z,e) mówi, że e $\in$ z; Podzbiór(z_1,z_2) mówi , że $z_1 \subset z_2$.
- Funkcje: Unia(z_1,z_2) – suma dwóch zbiorów: $z_1 \cup z_2$; Przecięcie(z_1,z_2) – iloczyn dwóch zbiorów: $z_1 \cap z_2$; Dołącz(z, x) – dodanie elementu x do zbioru z; Zbiór(x) – jest prawdziwe, gdy x jest zbiorem.

W logice pierwszego stopnia świat opisuje 8 reguł:

Reguła 1: Pozwala nam na wyodrębnienie ze świata elementów, które są zbiorami. Element jest albo zbiorem pustym, albo zbiorem, który jest utworzony z innego zbioru przez dodanie jednego elementu:

$\forall z$ Zbiór(z) $\Leftrightarrow$ (z=*ZbiórPusty* $\vee$ ($\exists x,z_2$ Zbiór(z_2) $\wedge$ z=Dołącz(z_2, x)))

Reguła 2: Zbiór pusty nie posiada elementów, czyli nie da się go utworzyć przez dodanie jakiegokolwiek elementu do innego zbioru:

$\neg\exists z,x$ Dołącz(z,x) = *ZbiórPusty*

Reguła 3: Dodanie do zbioru elementu, który już w nim jest niczego nie zmienia:

$\forall z,x$ Element(z,x) $\Leftrightarrow$ z=Dołącz(z,x)

Reguła 4: W zbiorze są tylko te elementy, które zostały do niego dołączone. Zbiór z jest utworzony przez ciąg operacji dołącz. Element x należy więc do zbioru, jeśli odwracając proces dołączania i odcinając rekurencyjnie kolejne elementy natrafimy w końcu na x.

$\forall z,x$ Element(z,x) $\Leftrightarrow \exists y,z_2$ (z=Dołącz(z_2,y) $\wedge$ (x=y $\vee$ Należy(x,z_2)))

Reguła 5: Pierwszy zbiór jest podzbiorem drugiego zbioru wtedy i tylko wtedy gdy każdy element należący do pierwszego zbioru należy również do drugiego zbioru:

$\forall z_1,z_2$ Podzbiór$(z_1,z_2) \Leftrightarrow (\forall x$ Element$(z_1,x) \Rightarrow$ Element(z_2,x))

Reguła 6: Dwa zbiory są równe wtedy i tylko wtedy gdy pierwszy jest podzbiorem drugiego, a drugi podzbiorem pierwszego.

$\forall z_1,z_2$ $(z_1 = z_2) \Leftrightarrow$ (Podzbiór$(z_1,z_2) \wedge$ Podzbiór(z_2,z_1))

Reguła 7: Element należy do przecięcia dwóch zbiorów wtedy i tylko wtedy gdy należy do pierwszego zbioru i należy do drugiego zbioru.

$\forall x,z_1,z_2$ Element(Przecięcie$(z_1,z_2),x) \Leftrightarrow$ (Element$(z_1,x) \wedge$ Element(z_2,x))

Reguła 8: Element należy do sumy dwóch zbiorów wtedy i tylko wtedy gdy należy do pierwszego zbioru lub należy do drugiego zbioru.

$\forall x,z_1,z_2$ Element(Suma$(z_1,z_2),x) \Leftrightarrow$ (Element$(z_1,x) \vee$ Element(z_2,x))

3.4 Logiczny agent w świecie wumpusa

Pierwszym etapem przy konstrukcji agenta w świecie wumpusa jest zdefiniowanie zależności pomiędzy otoczeniem i agentem. Zdanie, które będzie zawierało spostrzeżenia agenta musi być wyposażone w informację o czasie spostrzeżenia (czasem mogą być np. kolejne liczby naturalne). Mamy więc:

Spostrzeżenie([Odór, Przeciąg, Błysk, Nic, Nic], 3)

Ponadto akcjami agenta są:

Obrót(Lewo), Obrót(Prawo), Wprzód, Strzał, Podnieś, Zostaw, Wyjdź

W każdym momencie agent podejmuje jedną z tych 7 akcji.

Agent „krótkowzroczny”

Najprostszy agent podejmuje decyzje bazując tylko na swoich spostrzeżeniach tzn. gdy widzi np. błysk to wybiera akcję Podnieś itp. Dla takiego agenta najważniejsze jest spostrzeżenie w jednostkach czasu:

$\forall p,b,m,k,t$ Spostrzeżenie([Odór, p, b, m, k], t) $\Rightarrow$ Odór(t)

$\forall o,b,m,k,t$ Spostrzeżenie([o, Przeciąg, b, m, k], t) $\Rightarrow$ Przeciąg(t)

$\forall o,p,m,k,t$ Spostrzeżenie([o, p, Błysk, m, k], t) $\Rightarrow$ NaZłocie(t)

.....

na podstawie tych spostrzeżeń w kolejnych czasach agent podejmuje natychmiastowe akcje:

$\forall t$ NaZłocie(t) $\Rightarrow$ Akcja(Podnieś, t)

Mimo, iż agent „krótkowzroczny” jest stosunkowo prosty w implementacji, to jednak jego sukces w świecie wumpusa jest prawie niemożliwy. Dla samej operacji Wyjdź (powodującej wyjście z labiryntu, gdy agent jest w początkowej kratce) agent nie podejmuje właściwej decyzji, ponieważ ani fakt posiadania złota, ani aktualna lokalizacja agenta nie jest żadnym z jego spostrzeżeń.

Drugim problemem dla takiego agenta jest możliwość zapętlenia się. Gdy po jakimś czasie agent ten ponownie odwiedzi jakąś kratkę, wówczas wymienione powyżej reguły nakażą mu zachowywać się tak jak wcześniej i zacznie on powtarzać swoje czynności.

Reprezentowanie zmiany w świecie

Kolejnym etapem przy konstrukcji agenta w świecie wumpusa jest wprowadzenie pojęcia sytuacji. Pojedyncza sytuacja reprezentuje aktualny stan świata, i każda nawet najmniejsza jego zmiana w dowolnej chwili czasowej powoduje wygenerowanie nowej sytuacji. Ta metoda reprezentowania zmiany w świecie w logice pierwszego stopnia nosi nazwę *Situation Calculus*.

Weźmy dla przykładu świat przedstawiony na rysunku nr 2. Sytuacja początkowa oznaczana jest zawsze jako S_0. Pierwszą czynnością, jaką zrobił agent był obrót w prawo, co spowodowało, że świat znalazł się w sytuacji S_1. Następnie agent poszedł wprzód, a świat znalazł się w sytuacji S_2 itd.

By przedstawić dwa pierwsze położenia agenta w różnych sytuacjach użyjemy predykatu *Na(obiekt, kratka, sytuacja):*

$Na(Agent, [1,1], S_0) \wedge Na(Agent, [1,2], S_2)$

Możemy też użyć funkcji *Rezultat(Akcja, Sytuacja)* = *Sytuacja*, która da nam wynik:

$Rezultat(Obrót(Prawo), S_0) = S_1$

$Rezultat(Wprzód, S_1) = S_2$

.....

Rozwiążemy teraz problem, z którym nie mógł sobie poradzić agent „krótkowzroczny” mianowicie wyposażymy agenta w reguły pozwalające mu pamiętać, że niesie we sobą złoto lub nie. Potrzebny jest nam predykat *Przenośny(Obiekt)* , by określić które obiekty są przenośne, oraz predykat *Obecny(Obiekt, Sytuacja)* do określenia które obiekty agent spotka w różnych sytuacjach. Wykorzystamy także predykat *Trzyma(Obiekt, Sytuacja)* do określenia które obiekty są trzymane przez agenta w różnych sytuacjach.

Możemy teraz napisać reguły, dzięki którym agent będzie pamiętał, czy posiada złoto, czy nie:

Przenośny(Złoto)
$\forall$s NaZłocie(s) $\Rightarrow$ Obecny(Złoto, s)
$\forall$s,x Obecny(x,s) $\wedge$ Przenośny(x) $\Rightarrow$ Trzyma(x, Rezultat(Podnieś, s))
$\forall$s,x $\neg$Trzyma(x, Rezultat(Zostaw, s))

Do tego momentu mamy pewność, że agent podniesie złoto w pewnej sytuacji. Musimy jeszcze zagwarantować, by w następnych sytuacjach dalej je trzymał:

$$\forall s,x,a\ \text{Trzyma}(x, \text{Rezultat}(a,s)) \Leftrightarrow [(a=\text{Podnieś} \wedge \text{Obecny}(x,s) \wedge \text{Przenośny}(x)) \vee (\text{Trzyma}(x,s) \wedge a \neq \text{Zostaw})]$$

Pamiętanie lokalizacji

Przedstawiliśmy już predykat *Na*, który pozwalał zapamiętać w jakiej kratce znajduje się agent w kolejnych sytuacjach. By agent posiadał pełną informację o swojej lokalizacji i możliwościach poruszania się w swoim świecie potrzebna jest jeszcze:

- Orientacja agenta w kolejnych sytuacjach, która może przyjmować wartości: 0, 90, 180 lub 270 stopni np.:
  Orientacja(Agent, S_2) = 90
- Informacja o sąsiadujących kratkach na mapie. Wykorzystamy do tego predykat *KratkaPrzed(Lokalizacja, Orientacja) = Lokalizacja*.
  $\forall$x,y KratkaPrzed([x,y], 0) = [x, y+1]
  $\forall$x,y KratkaPrzed([x,y], 90) = [x+1, y]
  $\forall$x,y KratkaPrzed([x,y], 180) = [x, y-1]
  $\forall$x,y KratkaPrzed([x,y], 270) = [x-1, y]

Możemy teraz już konstruować poszczególne reguły logiczne, które będą opisywały ukryte zależności w świecie np.:
$\forall$x,y Odór(x,y) $\wedge$ Odór(x+1,y+1) $\Rightarrow$ Wumpus(x,y+1) $\vee$ Wumpus(x+1,y)

Strategia logicznego agenta

Agent w każdym momencie musi zdecydować się na jakąś akcję. Wiadome jest, że zanim znajdzie złoto będzie miał dylemat czy kontynuować poszukiwania, czy skierować się w stronę wyjścia. Logicznie działający agent powinien przed każdą czynnością robić listę akcji o największym priorytecie. Dla przykładu:

- najważniejszą akcją będzie podniesienie złota, gdy agent będzie się znajdował w kratce z błyskiem.
- Gdy agent będzie posiadał złoto, wówczas powinien podążać po bezpiecznych kwadratach w stronę wyjścia.
- Gdy agent nie posiada złota, ale wie, że istnieją nie odkryte przez niego kwadraty, które są bezpieczne, wówczas powinien do nich pójść.
- Gdy agent nie ma złota, nie ma też bezpiecznych kwadratów, które nie zostały jeszcze odwiedzone przez agenta, wówczas powinien on skierować się w stronę wyjścia.

Warto jeszcze zauważyć, że gdy w rozpatrywanym przez nas świecie Wumpus się porusza, wówczas agent powinien zapamiętywać ostatnio odwiedzona, bezpieczna kratkę i w razie wyczucia odoru wycofać się w to bezpieczne miejsce. Śmierć agenta jest bardzo kosztowna, więc ta czynność powinna posiadać największy priorytet.

3.5 Zadania

1. Przedstaw następujące zdania w logice pierwszego stopnia używając jednolitego języka (symboli, predykatów itp.) zdefiniowanego wcześniej:

a) Nie wszyscy studenci studiują jednocześnie Historię i Biologię.
b) Tylko jeden student nie zdał Historii.
c) Tylko jeden student nie zdał ani Historii ani Biologii.
d) Najlepszy wynik z Historii był lepszy niż najlepszy wynik z Biologii.
e) Każda osoba, która nie lubi wszystkich jaroszy jest mądra.
f) Nikt nie lubi mądrego jarosza.
g) Jest kobieta, która lubi wszystkich mężczyzn, którzy nie są jaroszami.
h) Jest fryzjer, który goli w mieście wszystkich, którzy sami się nie gola.
i) Żadna osoba nie lubi profesora, gdy nie jest on mądry.
j) Politycy mogą oszukiwać część ludzi przez cały czas, mogą oszukiwać wszystkich ludzi przez jakiś czas, ale nie mogą oszukiwać wszystkich ludzi przez cały czas.

6. Przedstaw w logice pierwszego stopnia zależności pomiędzy: bratem, siostrą, synem, córką, matką, ojcem, dziadkiem, babcią, kuzynem, kuzynką, ciotką i wujkiem.

7. Niech w świecie wumpusa znajduje się 2 niezależnych agentów nie mogących się komunikować. Każdy z nich usiłuje osobno dotrzeć do złota, i wygrywa ten, który znajdzie skarb jako pierwszy. W każdym takcie zegarowym agent może wykonać jakąś akcję; agenci nie mogą znajdować się w tej samej lokalizacji. Przy pomocy funkcji Jednocześnie(akcja1, akcja2) dającej w rezultacie połączone akcje 2 agentów zapisz zdania logiczne, które nie pozwolą, by agenci znaleźli się w tym samym polu.

8. Załóżmy, że agent posiadający m strzał znajduje się w świecie, w którym jest n wumpusów. Zapisz zdania, opisujące warunki i efekty zastosowania przez agenta akcji *Strzał*.

4. Budowanie Bazy Wiedzy

W rozdziale tym omówimy w jaki sposób budować bazę wiedzy opartą na logice pierwszego stopnia.

Przy procesie budowania Bazy Wiedzy najważniejsze są dwie czynności:

- Zebranie wszystkich reguł opisujących świat.
- Zapisanie tych reguł w poprawny sposób, który będzie umożliwiał rozbudowę bazy wiedzy.

Nie należy zapomnieć, że każda baza wiedzy jest ściśle powiązana z mechanizmem wnioskującym, dlatego reguły, które się w niej znajdują powinny być jasne, jednoznaczne, poprawne i efektywne.

Zajmiemy się w tym rozdziale właśnie efektywnością, a konkretnie sposobami zapisu różnego rodzaju informacji w logice pierwszego stopnia. Dla opanowania umiejętności projektowania efektywnych baz wiedzy nie wystarczy nam tylko mówić o tym zagadnieniu; potrzebujemy rozpatrzyć przykład. W tym celu przedstawimy najpierw źle (nieefektywnie) zbudowaną bazę wiedzy, a następnie tak ją zmodyfikujemy, by stała się efektywna.

4.1 Dobra i zła baza wiedzy

W związku z tym, że w każdej bazie wiedzy znajdują się reguły tworzące zdania logiczne będące odzwierciedleniem opisywanego świata, są one analizowana zarówno przez system wnioskujący, jak i projektanta bazy wiedzy. Możemy w tym miejscu założyć, że inwencja twórcza systemu wnioskującego jest ograniczona, ponieważ składa się na nią algorytm wnioskujący, który działa poprawnie. Pozostaje więc nam skupić się na inwencji twórczej projektanta systemu, od którego zależy efektywność budowanej bazy wiedzy.

Częstym błędem przy konstruowaniu bazy wiedzy jest stosowanie predykatów, które niosą ze sobą dużo informacji dla projektanta systemu, natomiast dla algorytmu wnioskującego są mało użyteczne.

Załóżmy, że w naszej bazie wiedzy znajdują się zdania dotyczące mechanicznych środków transportu na Ziemi. Niech w bazie wiedzy będzie się znajdował predykat:

AutoMającеBardzoSłabySilnik(Fiat126).

Dla człowieka powyższa informacja jest bardzo praktyczna, gdyż dowiaduje się on z niej, że auto Fiat126 posiada bardzo słaby silny.

Jednak system wnioskujący potraktuje powyższy predykat jak każdy inny, co spowoduje problemy przy dedukowaniu innych faktów. Wyposażmy naszą bazę wiedzy w regułę:

$\forall s\ AutoMającaBardzoSłabySilnik(s) \Rightarrow Wolny(s)$

, która o każdym samochodzie, posiadającym słaby silnik mówi, że jest on wolny. Gdybyśmy teraz chcieli zapisać inną regułę, tym razem dotyczącą aut o mocnym silniku, należałoby zapisać:

$\forall s\ AutoMającaMocnySilnik(s) \Rightarrow Szybki(s)$

Tu możemy zauważyć, że nasza baza wiedzy nie jest w porządku, ponieważ zmieniając tylko jeden parametr auta dostajemy nowy predykat, który nie ma dla systemu wnioskującego nic wspólnego z poprzednim. Co więcej, rozróżnienie silników bardzo słabych, słabych, średnich, mocnych i bardzo mocnych także tworzy całkowicie inne, nie mające ze sobą nic wspólnego predykaty.

Rozwiązaniem jest zastępowanie *BardzoDługichPredykatów* przez predykaty krótsze, posiadające jak najmniejszą ilość różnych informacji, które będzie można potem ze sobą scalić. W rezultacie by wyrazić wszystkie informacje o środkach transportu będziemy potrzebowali mniej reguł i predykatów. Spójrzmy w jaki sposób możemy zdefiniować auta marki Fiat i zależność pomiędzy ich silnikami, a prędkościami:

1. Fiat126 jest autem; Auta są środkami transportu; Środki transportu są Obiektami Fizycznymi, dlatego napiszemy:

   $Auto(Fiat126)$

   $\forall s\ Auto(s) \Rightarrow ŚrodekTransportu(s)$

   $\forall o\ ŚrodekTransportu(o) \Rightarrow ObiektFizyczny(o)$

   Tak zapisane reguły pozwalają nam na wyodrębnienie informacji dotyczących zarówno różnych aut jak i Środków Transportu.

2. „Fiat126 posiada bardzo słaby silnik” możemy zapisać:

   $WzględnaMoc(SilnikOd(Fiat126),SilnikOd(TypoweAuto))=Bardzo(Słaby)$

   Pozostaje jeszcze sprecyzować co oznacza Bardzo(Słaby) oraz czym jest obiekt TypoweAuto

3. By połączyć silniki z środkami transportu zapisujemy, że każdy środek transportu posiada silnik, który jest jego częścią:

   $\forall s\ ŚrodekTransportu(s) \Leftrightarrow Silnik(SilnikOd(s))$

   $\forall s\ JestCzęścią(SilnikOd(s), s)$

   SilnikOd(x) zwraca nam obiekt będący silnikiem, natomiast Silnik(y) zwraca prawdę lub falsz.

4. Jeśli coś jest częścią Obiektu Fizycznego, to:

   $\forall x,y\ JestCzęścią(x,y) \wedge ObiektFizyczny(y) \Rightarrow ObiektFizyczny(x)$

5. Środek transportu, który posiada słaby silnik w stosunku do typowego obiektu swojej kategorii, jest wolny.

$\forall s\ WzględnaMoc(SilnikOd(s), SilnikOd(TypowyObiekt(Typ(s)))) \leq Słaby \Rightarrow Wolny(s)$

$\forall a\ Auto(a) \Leftrightarrow Typ(a) = auto$

$TypoweAuto = TypowyObiekt(auto)$

6. Każdy silnik posiada swoją moc (silny, średni, słaby). Względna moc jest stosunkiem dwóch wielkości:

$\forall x\ Silnik(x) \Rightarrow \exists m\ Moc(x) = m$

$silny < średni < słaby$

$\forall x,y\ WzglednaMoc(x,y) = Moc(x) / Moc(y)$

7. Pozostaje jeszcze zdefiniować funkcję Bardzo. Średni jest neutralną wartością na skali:

$Średni = 1$

$\forall x\ \ x > Średni \Rightarrow Bardzo(x) > x$

$\forall x\ \ x < Średni \Rightarrow Bardzo(x) < x$

Tak zdefiniowane reguły i predykaty mimo iż są nie są najprostsze wprowadzają coś więcej niż tylko regułę dotyczącą aut o słabym silniku. Tworzą one zarysy hierarchii obiektów i ogólne zasady dotyczące wielkości. Ponadto reprezentują na tyle małe fakty, że mogą stać się elementami innych baz wiedzy, a ich scalanie jest stosunkowo proste.

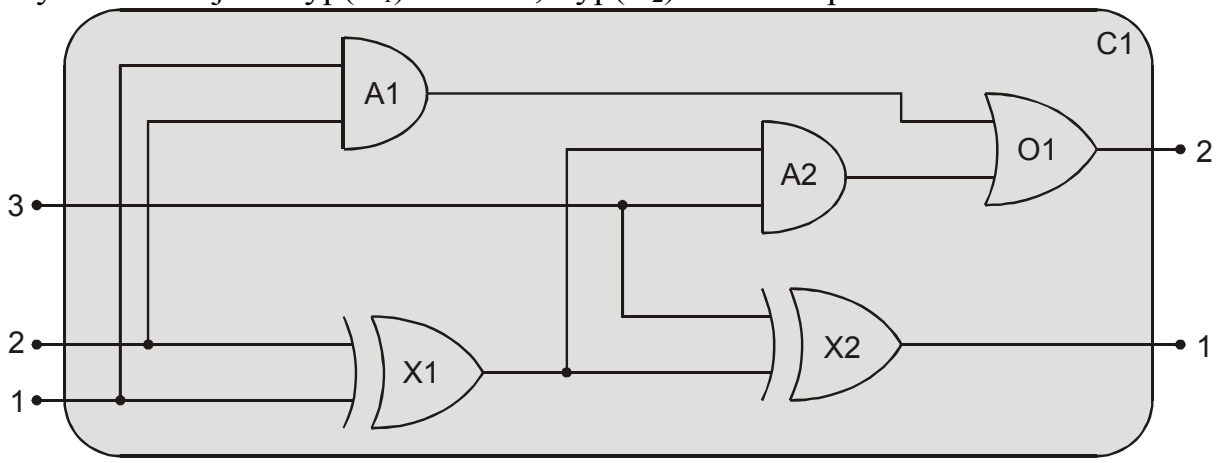
4.2 Baza Wiedzy dla układów scalonych

W tym paragrafie przedstawimy bazę wiedzy wykorzystywaną do analizowania zagadnień związanych z układami scalonymi. Rozpatrzymy układ scalony służący do dodawania dwóch bitów, razem z bitem przeniesienia, który przedstawiony jest na rysunku nr 5.

Zbudujemy w oparciu o ten schemat bazę wiedzy, która będzie potrafiła w dowolnym momencie podać jaka jest logiczna wartość na danym odcinku układu scalonego. Będziemy używali bramek AND, OR i XOR, które jak wiadomo posiadają dwa wejścia i jedno wyjście, które przyjmuje wartości zależne od sygnałów na wejściu.

Rozróżnimy więc w naszym modelu dwa podstawowe typy obiektów: sygnał oraz bramka. Pierwszym krokiem przy budowie takiej bazy wiedzy jest wybranie odpowiedniego języka.

Zaczniemy od podania typu poszczególnych bramek, co będzie odpowiadał predykat Typ. Dla bramek znajdujących się na rysunku nr 5 będziemy mieli kolejno: Typ(X_1) = XOR , Typ(A_2) = AND itp.



Rysunek 5 Układ scalony realizujący dodawanie

Teraz należy zająć się sygnałami w poszczególnych punktach obwodu. Musimy wiedzieć jaki sygnał znajduje się na każdym z wejść do każdej bramki, oraz na wyjściach wszystkich bramek. Moglibyśmy w tym celu użyć oznaczeń X_1Wej_1, X_1Wej_2 itp., ale spotkalibyśmy się wtedy z problemem długich, nieużytecznych dla systemu wnioskującego nazw, które omówione były w rozdziale 4.1, dlatego zamiast tego użyjemy dwóch funkcji: Wej i Wyj. Funkcje te będą miały postać: Wej(1, X_1), Wej(2, X_1) co reprezentować będzie sygnały odpowiednio na pierwszym wejściu bramki X_1, na drugim wejściu bramki X_1 itp. Funkcja Wyj będzie wyglądała podobnie.

W celu dodania do bazy wiedzy faktu, że jakieś dwa miejsca układu są ze sobą połączone użyjemy funkcji Połączone. By wyrazić że pierwsze (i jedyne!) wyjście bramki A_2 jest połączone z pierwszym wejściem bramki(O_1) napiszemy:

Połączone(Wyj(1,A_2), Wej(1,O_1))

Ostatecznie będziemy potrzebowali przypisać wartości sygnałom. Użyjemy w tym celu nazw *On* i *Off*. Dla „t” - konkretnego miejsca w sieci, Funkcja *Sygnał(t)* będzie zwracała wartość sygnału w tym miejscu.

Generalne zasady dotyczące układów scalonych

Na początku musimy w bazie wiedzy podać zasady, które funkcjonują dla wszystkich układów scalonych. Są to zależności pomiędzy sygnałami na wejściach bramek, a sygnałami na wyjściach bramek. Realizuje je 7 poniższych zasad:

1. Jeśli dwa punkty sieci są połączone, wówczas mają ten sam sygnał:

$\forall p_1, p_2$ *Połączone*$(p_1,p_2) \Rightarrow$ *Sygnał*(p_1) = *Sygnał*(p_2)

2. Dla jednego miejsca w sieci, sygnał jest albo *on*, albo *off*:
$\forall p\ (Sygnał(p)=On \vee Sygnał(p) = Off) \wedge (On \neq Off)$

3. Relacja *Połączone* jest symetryczna:
$\forall p_1,p_2\ \ Połączone(p_1,p_2) \Leftrightarrow Połączone(p_2,p_1)$

4. Na wyjściu bramki OR jest sygnał On wtedy i tylko wtedy gdy chociaż na jednym wejściu do tej bramki jest sygnał On:
$\forall b\ \ Typ(b)= OR \Rightarrow$
$Sygnał(Out(1,g)) = On \Leftrightarrow \exists n\ \ Sygnał(In(n, g)) = On$

5. Na wyjściu bramki XOR jest sygnał On wtedy i tylko wtedy gdy na wejściach do tej bramki są różne sygnały:
$\forall b\ \ Typ(b)= XOR \Rightarrow$
$Sygnał(Out(1,g)) = On \Leftrightarrow Sygnał(In(1, g)) \neq Sygnał(In(2, g))$

6. Na wyjściu bramki AND jest sygnał Off wtedy i tylko wtedy gdy chociaż na jednym wejściu do tej bramki jest sygnał Off:
$\forall b\ \ Typ(b)= AND \Rightarrow$
$Sygnał(Out(1,g)) = Off \Leftrightarrow \exists n\ \ Sygnał(In(n, g)) = Off$

7. Na wyjściu bramki NOT jest inny sygnał niż na jej wejściu.
$\forall b\ \ Typ(b)= NOT \Rightarrow Sygnał(Out(1,g)) \neq Sygnał(In(1,g))$

Specyficzne zasady omawianego układu

Na początku zapisujemy typ każdej bramki w układzie z rysunku nr 5
$Typ(X_1) = XOR$ $Typ(X_2) = XOR$
$Typ(A_1) = AND$ $Typ(A_2) = AND$
$Typ(O_1) = Or$

Teraz definiujemy schemat sieci przez zapisanie połączeń:

$Połączone(Wyj(1,X_1),Wej(1,X_2))$
$Połączone(Wyj(1,X_1),Wej(2,A_2))$
$Połączone(Wyj(1,A_2),Wej(1,O_1))$
$Połączone(Wyj(1,A_1),Wej(2,O_1))$
$Połączone(Wyj(1,X_2),Wyj(1,C_1))$
$Połączone(Wyj(1,O_1),Wyj(2,C_1))$
$Połączone(Wej(1,C_1),Wej(1,X_1)$
$Połączone(Wej(1,C_1),Wej(1,A_1)$
$Połączone(Wej(2,C_1),Wej(2,X_1)$
$Połączone(Wej(2,C_1),Wej(2,A_1)$
$Połączone(Wej(3,C_1),Wej(2,X_2)$
$Połączone(Wej(3,C_1),Wej(1,A_2)$

Dla tak zdefiniowanej bazy wiedzy możemy już zastosować algorytmy wnioskujące by dowiedzieć się np. dla jakich sygnałów wejściowych na wejściu układu, uzyskamy na wyjściu bit sumy = Off i bit przeniesienia = On.

$\exists i_1,i_2,i_3\ Sygnał(In(1,C_1)) = i_1 \wedge Sygnał(In(1,C_1)) = i_2 \wedge Sygnał(In(1,C_1)) = i_3 \wedge Sygnał(Out(1,C_1)) = Off \wedge Sygnał(Out(2,C_1)) = On$

Po tym zapytaniu uzyskamy oczekiwany rezultat:

$(i1=On \wedge i2=On \wedge i3=Off) \vee$
$(i1=On \wedge i2=Off \wedge i3=On) \vee$
$(i1=Off \wedge i2=On \wedge i3=On)$

4.3 Sposoby reprezentacji świata

W rozdziale 4.1 przedstawiliśmy źle skonstruowaną regułę należącą do bazy wiedzy, a następnie zamieniliśmy ją na szereg dobrych reguł. Świat który nas otacza okazuje się być jednak zdecydowanie bardziej skomplikowany niż prosta baza wiedzy dotycząca aut, a co za tym idzie trudniej jest go przedstawić w postaci reguł zapisanych w logice pierwszego stopnia.

W tym rozdziale podamy w jaki sposób zapisać bazę wiedzy, by mogła ona odzwierciedlać wszystko to, co się wokół nas dzieje. Skupimy się na następujących własnościach świata:

- Kategorie: Gdy mamy do czynienia z szeregiem obiektów posiadających pewną określoną cechę, to wygodnie jest przypisać im jakąś kategorię i cechę tą nadać tej kategorii. Np. kaseta i płyta są obie nośnikami danych i co za tym idzie, nadają się do zapisu muzyki.
- Miary: Wyrażając ilość zawsze musimy podać jednostkę w której jest ona wyrażona. Podamy sposoby rozróżnienia różnych wielkości w logice pierwszego stopnia.
- Obiekty złożone: Użytecznie jest przedstawiać obiekty jako elementy innych obiektów np. wszystkie województwa składają się na obiekt o nazwie Polska.
- Czas, Miejsce i Zmiana: Spotykamy się z tymi własnościami najczęściej. Każdy logiczny agent, który otrzymuje informacje z zewnątrz musi mieć zaimplementowane mechanizmy zmiany czasu i stanu. Zmiana miejsca jest nieodzowna dla agentów poruszających się po świecie.
- Wydarzenia i Procesy: Świat składa się z poszczególnych wydarzeń. W dalszej części tego rozdziału stworzymy agenta robiącego zakupy w supermarkecie. Zakupy podzielimy na wydarzenia np. szukanie towaru, zapłata za towar itp.

- Obiekty Fizyczne: Spotkaliśmy się już z nimi na początku rozdziału. Używamy ich, gdy mamy do czynienia z obiektami, które posiadają cechy fizyczne zmieniające się w czasie.
- Substancje: Trzeba odróżnić jedno jabłko od wody, soku jabłkowego itp. czyli substancji, których ilości nie znamy. Podamy w jaki sposób obiekty fizyczne i substancje się ze sobą łączą.
- Obiekty umysłowe i przekonania: Agent często będzie potrzebować wyciągać wnioski na podstawie własnej wiedzy, czyli będzie musiał wiedzieć jakie posiada reguły. To wymusza na agencie znajomość własnych przekonań i przekonań innych agentów znajdujących się w świecie

Reprezentowanie Kategorii

Z kategoriami spotkaliśmy się już przy rozpatrywaniu bazy wiedzy z autami. Zapisaliśmy wtedy Auto(a) jako zdanie prawdziwe, co oznaczało $a \in$ Auto czyli a jest autem.

Bardzo przydatną rzeczą jest identyfikowanie predykatów lub funkcji z obiektami je identyfikującymi. Chcielibyśmy przykładowo nadać jakąś cechę danej kategorii bez nadawania tej należącym do tej kategorii obiektom. W tym celu moglibyśmy predykat auto( ) utożsamić z obiektem Auta, a następnie zapisać: $Cena(TypowyObiekt(Auta)) = 30000$. Ta cecha zostałaby przypisana całej kategorii bez wyszczególniania należących do niej obiektów.

Obiekty te posiadałyby tą cechę dzięki *dziedziczeniu*, czyli przejmowaniu przez obiekty cech, które posiadają obiekty nadrzędne (w tym przypadku kategorie).

Logika pierwszego stopnia pozwala nam na zapis reguł dotyczących kategorii albo przy pomocy obiektu reprezentującego kategorię, albo przez nadanie tej samej własności wszystkim obiektom jednej kategorii.

- Obiekt należący do kategorii zapisujemy:
  $Fiat_{54} \in Fiaty$
- Kategoria jest podzbiorem większej kategorii:
  $Fiaty \subset Auta$
- Wszystkie obiekty jednej kategorii mają tą samą cechę:
  $\forall x \; x \in Fiaty \Rightarrow Słaby(x) \wedge Komfortowy(x)$
- Obiekty kategorii mogą być rozpoznane przez niektóre cechy:
  $\forall f \; (Słaby(f) \wedge Komfortowy(f)) \wedge f \in Kontyngent_{1999} \Rightarrow x \in Fiat$
- Kategorie jako całość posiadają swoje własności:
  $Fiaty \in ŚrodkiTransportu$

W ostatniej regule trzeba zauważyć, że ŚrodkiTransportu są zbiorem składającym się z kategorii.

Przy omawianiu kategorii spotykamy się jeszcze z trzema ważnymi pojęciami. Weźmy np. pory roku i miesiące. By wyrazić, że Wiosna i listopad są rozłączne, piszemy:

Rozłączne({Wiosna, Listopad})

By wyrazić, że rok rozkłada się w całości na pory roku i miesiące zapiszemy:

RozkładWyczerpujący({Miesiące$_{2\text{-}11}$,Zima},rok)

Jeśli obiekty wyczerpującego rozkładu są rozłączne, to mamy do czynienia z podziałem:

Podział({Wiosna, Lato, Jesień, Zima},rok)

Definicje tych trzech pojęć w logice pierwszego stopnia są następujące:

$\forall z \;\; Rozłączne(z) \Leftrightarrow (\forall a, b \;\; a \in z \wedge b \in z \wedge a \neq b \Rightarrow a \cap b = \varnothing)$

$\forall z,y \;\; RozkładWyczerpujący(z,y) \Leftrightarrow (\forall i \;\; i \in y \Leftrightarrow \exists a \;\; i \in a \;\wedge\; a \in z)$

$\forall z,y \;\; Podział(z,y) \Leftrightarrow Rozłączne(z) \wedge RozkładWyczerpujący(z,y)$

Miary

Zarówno w naturalnej jak i sztucznej reprezentacji świata, obiekty mogą mieć wielkość, masę koszt, temperaturę itp. Ogólnie, ilości są proste do reprezentacji. Weźmy np. koszt, który wynosi 4PLN. Wielkość ta jest równoważna 1$. Możemy zatem przez połączenie ilości z rzędem wielkości wyrazić dowolny koszt:

Koszt(Fiat126) = $(1000) = PLN(4000)

W tym przypadku wykorzystaliśmy funkcje „$" oraz „PLN"

Konwersja między jednostkami wygląda następująco:

$\forall d \; Centymetry(2.54 * d) = Cale(d)$

$\forall t \; Celsjusz(t) = Fahreneit(32+1.8 * t)$

Niektóre miary nie są numeryczne i by je porównać możemy użyć symbolu <. Spotkaliśmy się już z taką miarą, gdy określaliśmy moc silników. Dla wyrażenia ogólnej zasady, że auta Ferrari są Mocniejsze od auta Fiat napiszemy:

Moc(SilnikOd(TypowyObiekt(Ferrari)))>
Moc(SilnikOd(TypowyObiekt(Fiat)))

Obiekty Złożone

Idea reguły: jeden obiekt jest częścią drugiego, rozdział jest częścią książki itp. jest oczywista. Będziemy stosowali relację JestCzęścią by wyrazić

przynależność jednego obiektu do drugiego. Relacja JestCzęścią jest przechodnia oraz zwrotna:

JestCzęścią(Prokocim, Kraków)
JestCzęścią(Kraków, Polska)
JestCzęścią(Polska, Europa)

Z przechodniości wnioskujemy, że *JestCzęścią(Prokocim, Europa)*.

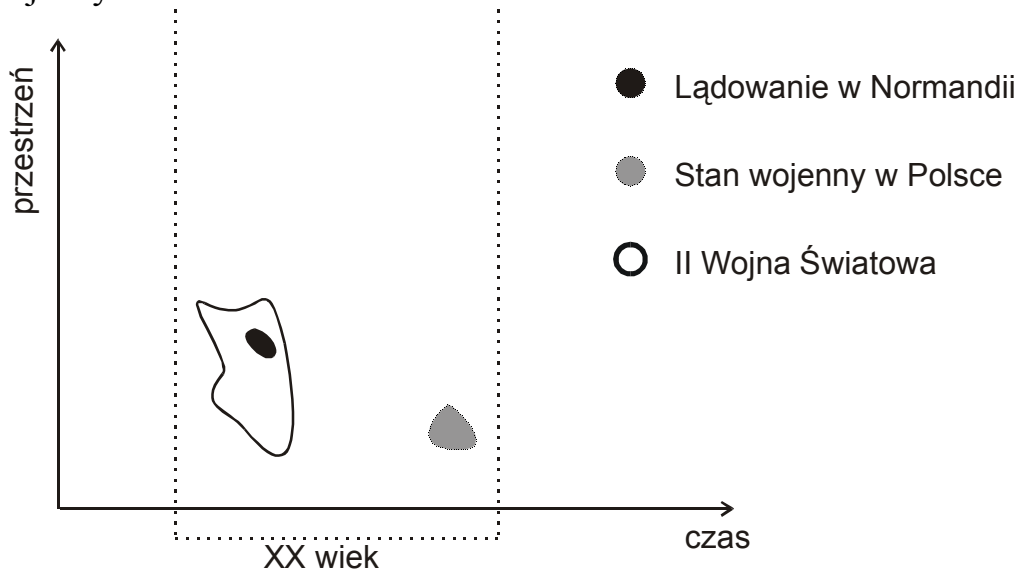
Każdy obiekt, który składa się z części nazywamy obiektem złożonym. Poniżej takim obiektem będzie Auto.

$\forall a\ Auto(a) \Rightarrow \exists k, z\ \ Kierownica(k) \wedge Zawieszenie(z)$
$\wedge JestCzęścią(k,a) \wedge JestCzęścią(z,a) \wedge$
$(k1\ Kierownica(k_1) \wedge JestCzęścią(k_1,a) \Rightarrow k_1 = k)$

Reprezentowanie zmiany przez zdarzenia

W logice zdań stosowaliśmy do reprezentacji zmian w świecie tzw. „Situation Calculus", które polegało na tym, że świat znajdował się w kolejnych stanach T_0, T_1, T_2, ... Ten sposób reprezentacji zmian w czasie nie jest jednak dobry, gdy mamy do czynienia z wieloma agentami, którzy w tej samej chwili wykonują jakąś czynność, lub gdy czasy wykonywania różnych czynności nie są takie same.

Zastosujemy więc inne podejście do zagadnienia zmian przestrzennych dokonujących się w czasie, nazywane „Event Calculus" (ang. Rachunkiem zdarzeń). Każde **wydarzenie** będziemy mogli zobrazować tak, jak na rysunku nr 6. Na osi poziomej jest czas, a na osi pionowej przestrzeń. Jeśli mówimy o wszystkich wydarzeniach w XX wieku, to będą one zawarte w obszarze ograniczonym dwoma datami:1900 i 2000. Wewnątrz tego obszaru znajdujemy pod-wydarzenia, jak np. II Wojna Światowa, lądowanie w Normandii, lub np. Stan wojenny w Polsce.



Rysunek 6 Wydarzenia w przestrzeni i czasie

Na rysunku nr 6 wydarzenie, które jest XX wiekiem nazywa się interwałem. Wydarzenie „II Wojna Światowa” jest pod-wydarzeniem w stosunku do II wieku itp. co zapisujemy:

PodWydarzenie(LądowanieWNormandii, IIWojnaŚwiatowa)
PodWydarzenie(IIWojnaŚwiatowa, XXWiek)
PodWydarzenie(StanWojennyWPolsce, XXWiek)

Wykorzystanie rachunku zdarzeń pozwala nam zatem na umieszczenie w jednym interwale dowolnej ilości wydarzeń. Mamy też możliwość umieszczania na osi czasu interwałów różnej długości.

Podobnie jak to miało miejsce z innymi obiektami, wydarzenia mogą być grupowane w kategorie. Gdy chcemy napisać, że w roku 1992 miała miejsce wojna na Bliskim Wschodzie, wtedy użyjemy formuły:
$\exists w \; w \in Wojny \wedge PodWydarzenie(w, 1992) \wedge$
JestCzęścią(Lokalizacja(w), BliskiWschód)

Miejsca

W logice pierwszego stopnia łatwo możemy też zapisać różnego rodzaju miejsca. Każde stałe miejsce na Ziemi jest niezmienne w czasie, dlatego użyjemy zamiast relacji PodWydarzenie, relacji „W” np.:

W(Kraków, Polska)

Miejsca posiadają różne kategorie, np. nie powinno się mylić miasta z planetą itp. dlatego dążąc do minimalizacji wprowadzamy relację Lokalizacja, która zwraca najmniejszy wycinek przestrzeni, w którym zaszło dane wydarzenie:

$\forall x,l \; Lokalizacja(x) = l \Leftrightarrow$
$NaObszarze(x,l) \wedge \forall l_2 \; NaObszarze(x,l_2) \Rightarrow W(l, l_2)$

Procesy

Dla uproszczenia podamy relację „Z(k,c)”, która będzie mówiła, że zdarzenie danej kategorii k zachodziło w interwale czasowym c, np.:

Z(Praca(Stefan), poniedziałek)

Co oznacza, że w poniedziałek Stefan pracował, nie wiemy jednak kiedy zaczął, ani kiedy skończył.

Gdy chcielibyśmy przedstawić wydarzenie, które trwało w ściśle określonych ramach czasowych, wówczas użyjemy relacji P:

P(Śniadanie(Stefan), 10-11)

Oznaczającej, że Stefan zaczął jeść śniadanie o 10, a skończył o 11.

Czasy, interwały i akcje

Zajmiemy się teraz czasem i sposobami reprezentacji momentów na osi czasu. Na początku musimy rozróżnić interwały, które są punktami na osi czasu, czyli które trwają 0, od interwałów, które zaczynają i kończą się w różnym czasie. Zapisujemy zatem następujące reguły:

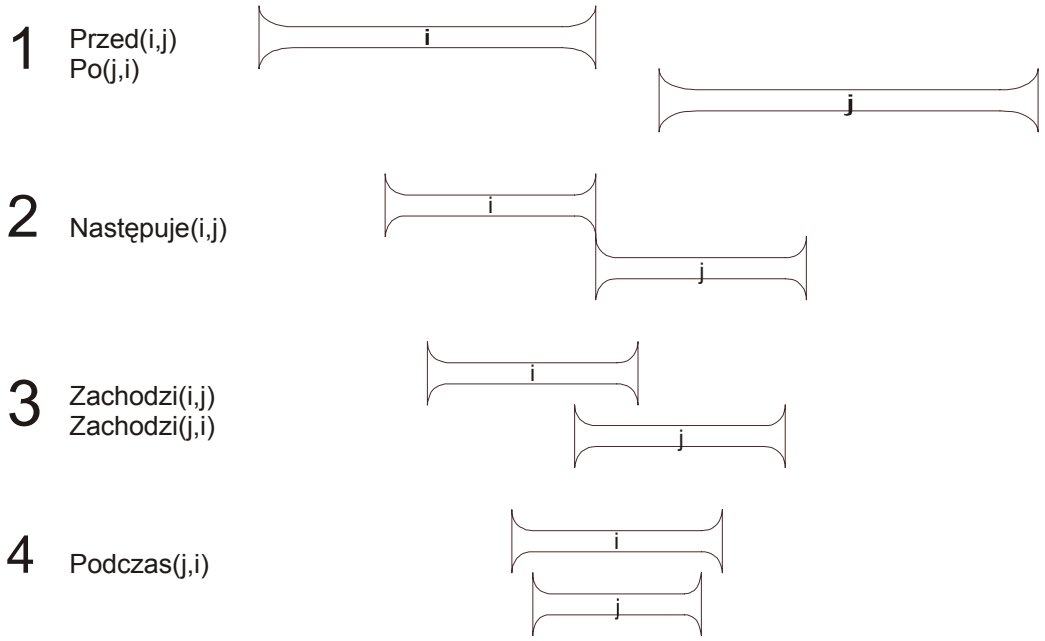
Podział({Momenty, DługieInterwały}, Interwały)

$\forall i \; i \in Interwały \Rightarrow (i \in Moment \Leftrightarrow CzasTrwania(i) = 0)$

, potrzebujemy jeszcze zdefiniować relację Długość:

$\forall i \; i \in Interwały \Rightarrow CzasTrwania(i) = Czas(Koniec(i)) - Czas(Start(i))$

Gdy mamy do czynienia z dwoma interwałami, to mogą one być rozmieszczone względem siebie na osi czasu na jeden z 4 sposobów, co przedstawia rysunek nr 7.



Rysunek 7 Rozmieszczenie interwałów czasowych

Dla przykładu zapiszmy regułę: Gdy 2 osoby biorą ślub, to są potem małżeństwem przez pewien czas począwszy od ślubu:

$\forall a,b,i \; P(Ślub(x,y), i) \Rightarrow \exists j \; P(Małżeństwo(x,y), j) \wedge Następuje(i,j)$

Wykorzystując przedstawiony mechanizm, możemy już stworzyć reguły, które opisują zmianę własności jakiegoś obiektu w czasie, np. zmianę ceny auta Fiat126 w pierwszych 3 miesiącach roku:

$P(Cena(Fiat126,\$(1000)),styczeń) \wedge P(Cena(Fiat126,\$(900)),luty) \wedge$
$P(Cena(Fiat126,\$(800)),marzec) \wedge Następuje(styczeń, luty) \wedge$
$Następuje(luty, marzec)$

Substancje

Z punktu widzenia konstruowanych przez nas baz wiedzy substancje będą posiadały jedną ważną cechę: Jeśli x będzie daną substancją, to nawet bardzo mała część tej substancji będzie posiadała takie same cechy, jak całość:

$\forall x,y \; x \in E98 \wedge JestCzęściąCałości(y,x) \Rightarrow y \in E98$

Obiekty umysłowe i przekonania

Skupimy się teraz na tym, co nasz sztuczny agent będzie wiedział o wiedzy posiadanej przez siebie oraz o wiedzy posiadanej przez innych agentów. Jest to bardzo duży krok do przodu, gdyż agent mogący korzystać z wiedzy zdobytej przez innych agentów poradzi sobie z większą ilością problemów dotąd przez niego nierozwiązywalnych.

Zastosujemy nową relację: *Wierzy(a, t)* , która będzie wyrażała, że agent a wierzy w term t. Pojawia się jednak od razu istotny problem. Jeśli term t będzie oznaczał to samo co term t_1, to „a" powinien także wierzyć w t_1, a wcześniej zapisana reguła tego wcale nie zapewnia. Innymi słowy:

$(t1 = t) \Rightarrow (Wierzy(a,t) \Leftrightarrow Wierzy(a,t_1))$

Wiemy jednak, że dany term może być zapisany na wiele sposobów, potrzeba nam więc uniwersalnego sposobu zapisu tego, w co agent wierzy. Wprowadzimy dwie nowe funkcje: pierwsza, którą oznaczymy Ozn(ciag) weźmie za argument ciąg znaków, który reprezentuje jakiś obiekt, a zwróci właśnie ten obiekt. Druga funkcja Naz(obiekt) weźmie za argument obiekt, a zwróci nazwę do tego obiektu. Przykładowo niech „Zbych" i „Zbyszek" będą odnosiły się do jednego obiektu: Zbigniew_Nowak, którego nazwiemy „ZN". Poniższe reguły łączą ze sobą trzy termy w jeden obiekt:

$Ozn(„Zbych") = Zbigniew\_Nowak \wedge Ozn(„Zbyszek") = Zbigniew\_Nowak$

$Naz(Zbigniew\_Nowak) = „ZN"$

Kolejnym krokiem jest dodanie do bazy wiedzy reguł, które zapewnią relacji Wierzy( ) przechodniość. Niech funkcja Kon(s_1, s_2, .., s_n) zwraca połączony ciąg znaków $s_1s_2...s_n$.

$\forall a,p,q \; Agent(a) \wedge Wierzy(a,p) \wedge Wierzy(a, Kon(p,"\Rightarrow",q)) \Rightarrow Wierzy(a,q)$

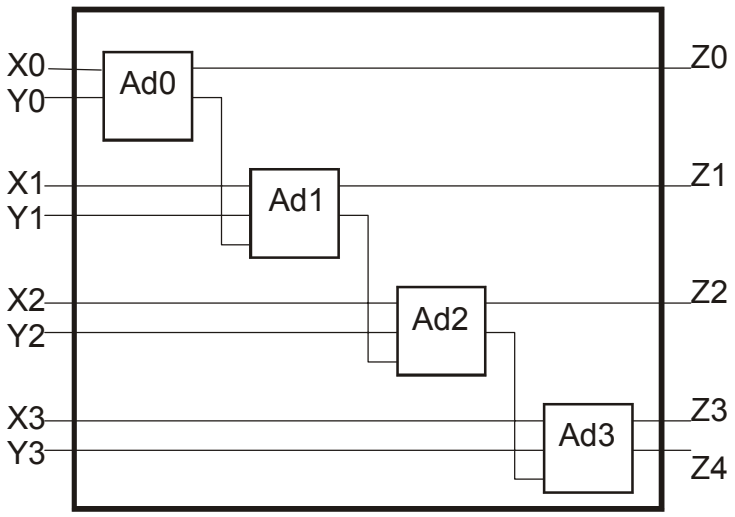
Teraz musimy jeszcze do bazy wiedzy dodać regułę mówiącą o tym, że nasz agent wierzy w swoje przekonania . W skrócie zapiszemy to tak:

$\forall a,p \; Agent(a) \wedge Wierzy(a,p) \Rightarrow Wierzy(a,"Wierzy(Naz(a), Naz(p))")$

4.4 Zadania

1. Skonstruuj takie zdania w logice pierwszego stopnia, które będą pozwalały na przeliczanie kursów walut (złotówka, marka, dolar) na podstawie dziennych wskaźników, które są dane.

2. Zapisz w logice pierwszego stopnia 4-bitowy sumator pokazany na rysunku nr 8, a następnie podaj zdania sprawdzające, czy działa on prawidłowo.



Rysunek 8 Schemat 4 bitowego sumatora

3. Zadaniem jest przeanalizowanie poniższego zdania logicznego:

„Wczoraj Jan poszedł do Krakowskiego magazynu Carrefour i kupił dwa kilogramy pomidorów i kilogram wołowiny”

i rozłożenie go na składniki logiki pierwszego stopnia w taki sposób, by dało się odpowiedzieć na następujące pytania:

a) Czy Jan kupił co najmniej 2 pomidory?
b) Czy Jan kupił mięso?
c) Czy w Krakowie jest magazyn Carrefour?
d) Jeśli Marta kupowałaby w tym samym czasie i w tym samym miejscu pomidory, czy Jan by ją zauważył?
e) Czy pomidory produkowane są w magazynie Carrefour?
f) Co Jan zrobi z pomidorami?
g) Czy magazyn Carrefour sprzedaje mydło?
h) Czy Jan wziął pieniądze lub kartę kredytową ze sobą?
i) Czy Jan ma mniej pieniędzy po wizycie w magazynie Carrefour?

5 Wnioskowania w logice pierwszego stopnia

W tym rozdziale podamy kompletny algorytm pozwalający na wnioskowanie nowych faktów na podstawie zdań zapisanych w logice pierwszego stopnia. Dla logiki zdań podaliśmy 7 głównych zasad, dzięki którym funkcjonował algorytm wnioskujący. Dla logiki pierwszego stopnia dodamy jeszcze 3 nowe reguły wnioskujące, służące głównie do pozbycia się kwantyfikatorów w zdaniach.

5.1 Reguły wnioskowania zawierające kwantyfikatory

Wcześniejsze reguły wnioskowania były stosunkowo oczywiste. Te które podamy poniżej pozwolą na pozbycie się kwantyfikatorów, które odtąd będą podstawiały do zdań logicznych zmienne. Wykorzystamy notację *ZAM(θ,α)* by wyrazić rezultat podstawienia do zdania α podmiany θ np.

ZAM({x/Adam, y/Aleksander}, Brat(x, y)) = Brat(Adam, Aleksander)

Trzema nowymi regułami są:

- **Eliminacja kwantyfikatora** ∀: Dla dowolnego zdania α, dowolnej zmiennej x, oraz dowolnego termu nie posiadającego zmiennych t:

  Dla $\forall x\ \alpha$ otrzymujemy $SUBST(\{x/t\}, \alpha)$

  Przykładowo dla zdania $\forall x\ Auto(x) \Rightarrow Kolor(x) = biały$ możemy zastosować podstawienie x/BBM2960 i dowiedzieć się, że auto o numerze rejestracyjnym BBM2960 jest białe.

- **Eliminacja Kwantyfikatora** ∃: Dla dowolnego zdania α oraz stałego symbolu s, który nie pojawia się w żadnej regule w bazie wiedzy:

  Dla $\exists x\ \alpha$ otrzymujemy $SUBST(\{x/s\}, \alpha)$

  Ważne jest, by stała s była unikatowa, tzn. by nie występowała w innych regułach. Możemy z nią utożsamiać jakiś nowy unikatowy alias. Przykładowo z $\exists x\ Auto(x) \wedge Kolor(x) = biały$ przez podstawienie x/A_{134} otrzymamy $Auto(A_{134}) \wedge Kolor(A_{134}) = biały$

- **Wprowadzenie Kwantyfikatora** ∃**:** Dla dowolnego zdania α, zmiennej x nie występującej w α oraz termu *t* nie zawierającego zmiennych:

  Dla α otrzymujemy $\exists x\ SUBST(\{x/t\}, \alpha)$

  Przykładowo ze zdania *Brat(Adam, Aleksander)* otrzymujemy $\exists x\ Brat(x, Aleksander)$.

5.2 Rozszerzona zasada Modus-Ponens

Przypomnijmy jak działała podstawowa zasada Modus-Ponens. Gdy wiedzieliśmy, że prawdziwe są zdania α oraz $\alpha \Rightarrow \beta$, to wnioskowaliśmy, że β jest zdaniem prawdziwym.

Weźmy teraz przykładową bazę wiedzy zawierającą zdania:

Auto(Fiat126) (1)

Posiada(Adam,Fiat126) (2)

$\forall a\ Auto(a) \wedge Posiada(Adam, a) \Rightarrow Dociera(Adam, a)$ (3)

Standardowo, by wywnioskować, że Adam dociera Fiata126 musielibyśmy najpierw w (3) zdaniu zastosować eliminację kwantyfikatora $\forall$, otrzymując:

$Auto(x) \wedge Posiada(Adam, x) \Rightarrow Dociera(Adam,x)$ (4)

następnie połączyć ze sobą zdania (1) i (2) otrzymując:

$Auto(Fiat126) \wedge Posiada(Adam,Fiat126)$

Podstawić w zdaniu(4) za x/Fiat126 i zastosować zasadę Modus-Ponens:

Dociera(Adam,Fiat126)

Przedstawiona poniżej rozszerzona zasada Modus-Ponens wykonuje wszystkie te czynności w jednym kroku:

Rozszerzona zasada ModusPonens: Dla zdań atomowych p_i, p_i' oraz q, takich, że $\forall i\ ZAM(\theta,p_i) = ZAM(\theta,p_i')$:

Mając zdania: $p1', p2', ..., pn', (p1 \wedge p2 \wedge ... \wedge pn \Rightarrow q)$

Wnioskujemy: $ZAM(\theta, q)$

W powyższej regule spotykamy n+1 zdań, przy pomocy których generujemy jedno zdanie $ZAM(\theta, q)$.

Dla przykładu weźmy bazę wiedzy podaną powyżej i zastosujmy do niej podaną zasadę; n = 2:

p_1'	jest Auto(Fiat126)	p_1	jest Auto(a)
p_2'	jest Posiada(Adam,Fiat126)	p_2	jest Posiada(Adam, a)
θ	jest {a/Fiat126}	q	jest Dociera(Adam, a)

Wniosek: $ZAM(\theta, q)$ daje *Dociera(Adam, Fiat126)*

Rozszerzona zasada Modus-Ponens jest efektywna, gdyż:

1) Algorytm wnioskujący wykonuje większe kroki składając małe reguły w większe
2) Podstawienia, jak przekonamy się później, nie są dokonywane przypadkowo. Jeśli istnieje odpowiednie podstawienie, to algorytm **unifikacji** je znajdzie.
3) Wykorzystujemy zdania zapisane w **formie kanonicznej** w bazie wiedzy i generujemy nowe zdania zapisane w tej formie.

Forma Kanoniczna

Jak wiadomo by nasz mechanizm wnioskujący działał prawidłowo, potrzeba aby wszystkie zdania w bazie wiedzy podchodziły pod schemat rozszerzonej reguły Modus-Ponens. Innymi słowy każde zdanie musi być implikacją, w której poprzednik implikacji jest koniunkcją zdań atomowych, a następnik implikacji pojedynczym atomem. Takie zdania nazywane są Klauzulami Horna, a baza wiedzy zawierająca same klauzule Horna jest w postaci normalnej Horna.

Gdy dodajemy do bazy wiedzy nowe zdanie, to może być ono przekształcone do klauzuli Horna. Na początku pozbywamy się kwantyfikatora $\exists$ i rozbijamy zdanie na zdania atomowe np. $\exists x\ Auto(x) \wedge Posiada(Adam, x)$ zostanie zamienione na zdania: $Auto(A1) \wedge Posiada(Adam, A1)$, gdzie A1 jest już konkretnym autem.

Teraz pozbywamy się kwantyfikatorów $\forall$ przez podstawienie do zdań zmiennych. Zdanie $\forall y\ \ Posiada(y, A1)$ przyjmie postać $Posiada(y,A1)$ przy czym y pozostanie zmienną. Powstaną nam zatem nowe Klauzule Horna.

Unifikacja

Nieodłączną częścią algorytmu wnioskującego jest funkcja UNIFY, która dla dwóch zdań atomowych p oraz q znajduje takie zamianę zmiennych, by po tej zamianie zdania te wyglądały na takie same. Funkcja UNIFY bierze więc jako parametry dwa zdania, a zwraca listę podstawień θ, tzw. **Unifikator**. Gdy nie istnieje żadne podstawienie, UNIFY zwróci **fail**. Formalnie:

$UNIFY(p,q) = \theta \Leftrightarrow ZAM(\theta, p) = ZAM(\theta, q)$

Kod źródłowy tej funkcji przedstawimy w rozdziale 5.5, gdyż nie jest on najłatwiejszy. Przedstawimy teraz znajdowanie rozwiązań przez funkcję UNIFY. Niech w naszej bazie wiedzy będzie reguła:

$Zna(Adam, x) \Rightarrow Lubi(Adam, x)$

mówiąca, że Adam lubi wszystkie osoby, które zna. Będziemy się teraz chcieli dowiedzieć, które osoby Adam lubi, gdy w bazie wiedzy będą następujące zdania:

Zna(Adam, Zbyszek)
Zna(z, Matka(z))
Zna(y, Marta)
Zna(x, Ewa)

Stosując funkcję UNIFY otrzymamy następujące rezultaty:

UNIFY(ZNA(Adam,x), ZNA(Adam,Zbyszek)) = {x/Zbyszek}
UNIFY(ZNA(Adam,x), ZNA(z,Matka(z))) = {z/Adam, x/Matka(Adam)}
UNIFY(ZNA(Adam,x), ZNA(y,Marta)) = {y/Adam, x/Marta}
UNIFY(ZNA(Adam,x), ZNA(x,Ewa)) = fail

Ostatnim rezultatem jest fail, ponieważ niemożliwe jest, by zmienna x była naraz Adamem i Ewą.

Stad od razu widzimy, że Adam lubi Zbyszka, swoją Matkę i Martę. Kolejnym problemem jest to, że jeśli istnieje jakieś podstawienie, to jest ich od razu nieskończenie wiele. Np.:

UNIFY(ZNA(Adam,x), ZNA(Adam,Zbyszek)) = {x/Zbyszek} ∨ { x/Zbyszek, w/Marek} ∨ { x/Zbyszek , k/Krzysztof} ∨...

Dodając nieistotne podstawienia nie zburzymy zgodności zdań, a lista θ będzie się wydłużała w nieskończoność. Za rezultat działania funkcji UNIFY będziemy więc uważali najkrótsze prawidłowe podstawienie.

5.3 Wnioskowanie w przód i w tył

Teraz, gdy mamy już poprawnie skonstruowaną bazę wiedzy i dysponujemy rozszerzoną zasadą Modus-Ponens możemy przedstawić dwa algorytmy wnioskujące: wnioskowanie w przód i wnioskowanie w tył.

Wnioskowanie wprzód jest zwykle używane, gdy dodawane jest jakieś zdanie do bazy wiedzy. Wówczas sprawdzane jest, czy przy pomocy tego nowego zdania nie da się wygenerować jakiegoś nowego zdania. Sprawdzane są przy tym wszystkie implikacje i często powstaje dużo nowych zdań, które są nieistotne.

Wnioskowanie w tył stosujemy wtedy, gdy chcemy dowieść jakieś konkretne zdanie. Jeśli tego zdania nie ma w bazie wiedzy, wówczas sprawdzamy, czy może być ono wywnioskowane z jakiegoś innego zdania i w razie potrzeby sprawdzany czy to inne zdanie jest prawdziwe itp.

Wnioskowanie w przód

Wnioskowanie w przód zaczyna działać, gdy do bazy wiedzy dodajemy nowe zdanie np. p. Celem algorytmu jest znalezienie w bazie wiedzy takich implikacji, dla których p jest poprzednikiem. Jeśli w takiej implikacji istnieją oprócz p inne zdania będące poprzednikami i jeśli wiemy, że te zdania są prawdziwe, wtedy możemy dodać do bazy wiedzy następnik implikacji i uruchomić algorytm od nowa (tym razem dodawanym zdaniem jest następnik implikacji).

Jeśli nie możemy stwierdzić o wszystkich poprzednikach danej implikacji, że są prawdziwe, wtedy rekurencyjnie szukamy zamiany dla kolejnych nieznanych poprzedników, powiększając przy tym listę zmiennych do zamiany.

Funkcja WNIOSK_W_PRZOD wykorzystuje także ideę **zmienników**. Jedno zdanie jest zmiennikiem dla drugiego, jeśli różnią się one między sobą tylko nazwami zmiennych w tych samych miejscach zdań. Przykładowo

Lubi(x, Ewa) oraz Lubi(y, Ewa) są zmiennikami, gdyż x $\leftrightarrow$ y. Zmiennikami nie będą jednak zdania: Lubi(x, x) i Lubi(x, y), gdyż nie są to te same zadania.

Wprowadzimy jeszcze pojęcie złożenia, którą wykorzystamy do składania listy zamian. UNIFY(θ_1,θ_2) jest zamianą, tzn. najpierw do zdania aplikujemy zamianę θ_1, a następnie θ_2. Innymi słowy:

$ZAM(UNIFY(\theta_1,\theta_2), \alpha) = ZAM(\theta_2 , ZAM(\theta_1, \alpha))$

Poniżej przedstawiony jest omawiany algorytm (KB – baza wiedzy)

```
1  procedure WNIOSK_W_PRZOD(KB, p)
2      if  w KB jest zdanie będące zmiennikiem p then
3          return
4      Dodaj p do KB
5      for each (p1∧...∧pn ⇒ q) w KB, gdy ∃i, dla którego
6      istnieje unifikator UNIFY(pi,p) = θ
7      do
8          WNIOSK_DALEJ(KB,[p1,...,pi-1,pi+1,pn],q,θ)
9      end
```

```
1  procedure WNIOSK_DALEJ(KB, poprzednicy, następnik, θ)
2      if  poprzednicy = [] then
3          WNIOSK_W_PRZOD(KB, ZAM(θ, następnik))
4      else for each zdania p' w KB takiego, że
5      UNIFY(p', ZAM(θ, PIERWSZY(poprzednicy))) = θ2
6      do
7          WNIOSK_DALEJ(KB, RESZTA(poprzednicy),
8          Następnik, UNIFY(θ, θ2))
9      end
```

Działanie algorytmu jest następujące. Każde nowe zdanie, które przychodzi do bazy wiedzy powoduje wywołanie procedury WNIOSK_W_PRZOD. Jeśli zdanie to już jest w bazie wiedzy, to algorytm procedura (jedna z gałęzi algorytmu) kończy działanie (linia nr 3). Pętla w linii 5 szuka takiej implikacji, dla której p jest jednym z poprzedników. Jeśli istnieją inni poprzednicy, wówczas wywoływana jest procedura WNIOSK_DALEJ, która szuka rekurencyjnie zamiany dla kolejnych poprzedników. Gdy lista poprzedników jest już pusta, a algorytm jest w linii 3 procedury WNIOSK_DALEJ, wtedy następnik jest dowiedziony, a algorytm dodaje go do bazy wiedzy w linii 3 procedury WNIOSK_W_PRZOD.

Wnioskowanie w tył

Wnioskowanie w przód służy do tego, by odpowiadać na wszystkie pytania stawiane bazie wiedzy.

Algorytm na początku szuka w bazie wiedzy odpowiedzi na stawiane mu pytanie i jeśli jej nie znajdzie posuwa się w tył, czyli analizuje kolejne implikacje od celu, aż do źródła, którym w tym przypadku są zdania w bazie wiedzy, dla których możemy ustalić wartość logiczną. W poniższym algorytmie pytamy: czy prawdziwe jest zdanie q?

function WNIOSK_W_TYŁ(KB, q) **return** unifikator
    WNIOSK_LISTA(KB,[q],{})

function WNIOSK_LIST(KB, list, θ) **return** unifikator
wejście: *KB – baza wiedzy*
list – lista zdań o które się pytamy (θ już zaaplikowane)
θ – aktualny unifikator
zmienne lokalne: podst – zbiór unifikatorów, początkowo pusty
(zbiór lokalnie możliwych podstawień)

if list = [] **then return** θ
q ←PIERWSZY(list)
for each q_i' w KB , takiego że spełnione jest $\theta_i = UNIFY(q,q_i')$ do
    podst ← podst ∪ UNIFY (θ,θ_i)
end
for each zdania $(p_1 \wedge \ldots \wedge p_n \Rightarrow q_i')$ w KB takiego,
że spełnione jest $\theta_i = UNIFY(q,q_i')$
do
    podst←WNIOSK_LIST(KB,ZAM(θ_i,[p_1,...,p_n]),
                UNIFY $(\theta,\theta_i)) \cup$ podst
end
return WNIOSK_LIST(KB, ZAM(Q_i,RESZTA(list)),Q_i) dla
    każdego unifikatora Q_i ze zbioru podst.

Jak widać powyżej, algorytm zaczyna pracę otrzymując na wejściu tylko jedno zdanie: q, o które się pytamy. Na początku stara się on znaleźć takie możliwe podstawienie, by zdanie q wynikało bezpośrednio z bazy wiedzy. Widzimy w tym momencie, że takich zdań może być dużo, dlatego możliwe podstawienia zapisywane są w zmiennej lokalnej *podst*.

Teraz algorytm szuka w bazie wiedzy implikacji, z których wynika zdanie q. By ta implikacja był spełniona, każdy poprzednik w tej implikacji musi być spełniony, dlatego jeśli nie wiemy czy poprzednik implikacji jest spełniony, uruchamiamy dla niego rekurencyjnie procedurę WNIOSK_W_TYŁ.

Po zakończeniu tego wywołania rekurencyjnego zbiór możliwych podstawień reprezentowany przez zmienną podst, zostanie odpowiednio zmodyfikowany.

Ostatnim krokiem algorytmu jest wywołanie przez niego samego siebie dla wszystkich niewiadomych zdań, przy każdym z możliwych podstawień ze zbioru podst. Jeśli algorytm znalazł jakieś podstawienie (czyli zdanie q może być prawdziwe) to przejdzie dalej, jeśli natomiast q nie jest zdaniem prawdziwym, wówczas algorytm zwróci fałsz.

5.4 Ostateczna procedura wnioskowania

Pokażemy teraz, że przedstawiona wcześniej procedura nie jest w stanie poradzić sobie z wszystkimi rodzajami baz wiedzy zapisanych w logice pierwszego stopnia. Rozpatrzmy następujące zdania:

$\forall x\ A(x) \Rightarrow B(x)$
$\forall x\ \neg A(x) \Rightarrow C(x)$
$\forall x\ B(x) \Rightarrow D(x)$
$\forall x\ C(x) \Rightarrow D(x)$

Widzimy, że prawdziwe jest zdanie D(x), jednak przedstawione dotychczas algorytmy nie poradzą sobie z tym problemem, gdyż zdanie $\forall x\ \neg A(x) \Rightarrow C(x)$ nie może być przedstawione w postaci Horna. Ten fakt mówi nam, że musimy powiększyć naszą listę reguł wnioskowania, by dało się na jej podstawie wydedukować wszystkie możliwe reguły.

Dysponujemy już prostą regułą rozdzielającą dla logiki zdań, która wygląda następująco:

Dla $\alpha \vee \beta,\ \neg\beta \vee \delta$ otrzymujemy $\alpha \vee \delta$.

,lub dla implikacji:

dla $\neg\alpha \Rightarrow \beta,\ \beta \Rightarrow \delta$ otrzymujemy $\neg\alpha \Rightarrow \delta$

W powyższej regule implikacje posiadają dokładnie 2 składniki. Możemy tą regułę tak rozszerzyć na więcej składników:

- **Rozszerzona reguła rozdzielająca (alternatywy):**

Dla zdań p_i oraz q_i, takich że UNIFY $(p_j, \neg q_k) = \theta$.

Dla zdań: $p_1 \vee \ldots \vee p_j \vee \ldots \vee p_m$ oraz $q_1 \vee \ldots \vee q_k \vee \ldots \vee q_n$ otrzymujemy:
$ZAM(\theta,(p_1 \vee \ldots \vee p_{j-1} \vee p_{j+1} \vee \ldots p_m \vee p_1 \vee \ldots \vee p_{k-1} \vee p_{k+1} \vee \ldots p_n))$

- **Rozszerzona reguła rozdzielająca (implikacje):**

Dla atomów p_i, q_i, r_i, s_i takich, że UNIFY $(p_j,q_k) = \theta$.
Dla zdań: $p_1 \wedge \ldots \wedge p_j \wedge \ldots \wedge p_{n1} \Rightarrow r_1 \vee \ldots r_{n2}$ oraz $s_1 \wedge \ldots \wedge s_{n3} \Rightarrow q_1 \vee \ldots \vee q_k \vee \ldots \vee q_{n4}$
Otrzymujemy:
$ZAM(\theta,p_1 \wedge \ldots \wedge p_{j-1} \wedge p_{j+1} \wedge \ldots \wedge p_{n1} \wedge s_1 \wedge \ldots \wedge s_{n3} \Rightarrow r_1 \vee \ldots \vee r_{n2} \vee q_1 \vee \ldots \vee_{qk-1} \vee q_{k+1} \vee \ldots \vee q_{n4}))$

Sprowadzanie do postaci normalnej

Pokażemy teraz jak sprowadzić do postaci normalnej dowolne zdanie z bazy wiedzy zapisane w logice pierwszego stopnia. Zdanie takie będzie już akceptowane przez rozszerzone reguły rozdzielające:

- Eliminacja implikacji: Zamieniamy wszystkie implikacje $p \Rightarrow q$ na alternatywę $\neg p \vee q$.
- Wstawienie $\neg$ „do środka": Korzystając z Praw de Morgana'a wstawiamy negacje do środka wyrażenia:
  $\neg(p \vee q)$ zamiana na $\neg p \wedge \neg q$ $\neg(p \wedge q)$ zamiana na $\neg p \vee \neg q$
  $\neg \forall x\ q$ zamiana na $\exists x\ \neg q$ $\neg \exists x\ q$ zamiana na $\forall x\ \neg q$
  oraz eliminacja podwójnej negacji: $\neg\neg q$ zamiana na q
- Standaryzacja zmiennych: Dla zdań takich jak: $\forall x\ Q(x) \vee \exists x\ F(x)$ zamieniamy nazwy zmiennych by uniknąć nieporozumień, gdy później pozbędziemy się kwantyfikatorów.
- Przesunięcie kwantyfikatorów na lewo: Bez zmiany sensu zdania przesuwamy na jego początek kolejne kwantyfikatory np.:
  $p \vee \forall x\ q$ zamieniamy na $\forall x\ p \vee q$.
- Eliminacja kwantyfikatora $\exists$: Tak jak w rozdziale 5.1 Wprowadzamy zamiast kwantyfikatora odpowiednią stałą. Jeśli zmienna stojąca przy kwantyfikatorze zależy od innej zmiennej, wówczas stosujemy funkcję, która nie pojawia się nigdzie indziej w bazie wiedzy:
  $\forall x\ Q(x) \Rightarrow \exists y\ K(y) \vee G(x,y)$ zamieniamy na:
  $\forall x\ Q(x) \Rightarrow K(F(x)) \vee G(x,F(x))$, gdzie „F" jest funkcją.
- Rozdzielenie $\wedge$ na $\vee$. $(a \wedge b) \vee c$ zamieniamy na $(a \vee c) \wedge (b \vee c)$
- Łączenie koniunkcji i alternatyw: $(a \wedge b) \wedge c$ zamieniamy na $(a \wedge b \wedge c)$, $(a \vee b) \vee c$ zamieniamy na $(a \vee b \vee c)$.

Pozostaje jeszcze rozpatrzyć ważny typ zdań, które zawierają w sobie znak równości np:

$\forall x\ K(x) = G(x)$

w takim przypadku stosujemy regułę demodulacji, która wygląda następująco:

- Dla dowolnych termów x,y,z gdzie UNIFY(x,z) = θ .

x=y, to zdanie (.... z ...) zamieniamy na (... ZAM(θ,y) ...)

5.5 Procedura Unifikacyjna

Przedstawimy teraz kod funkcji unifikacyjnej, którą stosowaliśmy w algorytmach wnioskowania. Wykorzystamy 2 funkcje rozbijające zdania na operatory i argumenty: Dla zdania „z" postaci P(x) ∨ Q(x) będziemy mieli będziemy mieli: OP(z) = [∨] i ARG(z) = [P(x), Q(x)].

Funkcja unifikacyjna działa rekurencyjnie i próbuje dopasować 2 wyrażenia symbol po symbolu:

```
function UNIFY(x,y) returns podstawienie, by x i y były identyczne,
                                                    jeśli to jest możliwe
UNIFY_PART(x,y,{})
```

```
function UNIFY_PART(x,y,θ) returns podstawienie, by x i y były
                                                    identyczne
inputs: x,y – zmienna, stała, lista argumentów, wyrażenie złożone
        θ - już zbudowane, częściowe podstawienie

if θ = błąd then return błąd
else if x=y then return θ
else if ZMIENNA?(x) then return UNIFY_ZM(x,y, θ)
else if ZMIENNA?(y) then return UNIFY_ZM(y,x, θ)
else if ZŁOŻONE?(x) and ZŁOŻONE?(y) then
     return UNIFY_PART(ARG(x),ARG(y),UNIFY_PART(OP(x),OP(y),θ)
else if LISTA?(x) and LISTA?(y) then
     return UNIFY_PART(RESZTA(x),RESZTA(y),UNIFY_PART(
                                   PIERWSZY(x), PIERWSZY(y),θ))
```

```
function UNIFY_ZM(zm, x, θ) returns podstawienie
inputs:    zm – zmienna, x – dowolne wyrażenie,
           θ - już zbudowane, częściowe podstawienie

if {zm/wyraż} ∈θ then return UNIFY_PART(wyraż, x, θ)
else if {x/wyraż} ∈θ then return UNIFY_PART(zm, wyraż, θ)
else if zm występuje gdzieś w x then return błąd
else return dodaj podstawienie {zm/x} do θ
```

Funkcja UNIFY_PART stara się dopasować do siebie część początkowych wyrażeń x i y. Jeśli wyrażenie są takie same, to nie ma potrzeby szukania podstawienia. Jeśli wyrażenie jest złożone, wtedy na początku analizowana jest zgodność list ich operatorów, a następnie zgodność argumentów.

Za analizę zgodność 2 argumentów jest odpowiedzialna funkcja UNIFY_ZM, która na początku sprawdza, czy podstawienie, które uczyniłoby 2 argumenty identyczne nie znajduje się już w zbiorze podstawień. Teraz sprawdzane jest, czy jeden argument nie jest częścią drugiego argumentu, co świadczyłoby o niezgodności argumentów. Jeśli żaden z powyższych warunków nie był spełniony, to dodawane jest do zbioru θ nowe podstawienie.

5.6 Indeksowanie bazy wiedzy

Samo wykorzystanie w procedurze wnioskującej efektywnego algorytmu nie jest gwarantem szybkiego wnioskowania faktów wynikających z bazy wiedzy. Zauważmy, że za każdym razem algorytm wnioskujący musi przeglądać bazę wiedzy w poszukiwaniu określonego zdania. Dla baz wiedzy zawierających n zdań, czas takiego jednego przeszukiwania byłby O(n), co jest wielkością stosunkowo dużą.

Dobrym sposobem na ten problem jest zastosowanie tablicy haszującej dla bazy wiedzy. Warto na początku budować w miarę proste zdania logiczne, a potem odpowiednie predykaty umieszczać w kontenerach tablicy haszującej. Algorytm przeszukujący bazę wiedzy w celu znalezienia jakiegoś zdania, najpierw ustala jakie predykaty są w tym zdaniu, a następnie sprawdza tylko te kontenery, które są związane z tymi predykatami.

W kontenerze z danym predykatem mamy 4 kolumny zawierające kolejno:

- Prawdziwe zdania z tym predykatem
- Fałszywe zdania z tym predykatem
- Implikacje, w których predykat jest jednym z poprzedników
- Implikacje, w których predykat jest wnioskiem

Weźmy dla przykładu predykat Brat(x,y). Kontener zawierający ten predykat może wyglądać następująco:

Brat	*Brat(Adam, Aleksander)* *Brat(Adam, Ewa)*	*¬Brat(Ewa, Adam)*	*Brat(x,y)∧Mężczyzna(y) ⇒ Brat(y,x)*	*Brat(x,y) ⇒ Mężczyzna(x)*

5.7 Zadania

1. Dla każdej z par zdań, podaj najbardziej generalną unifikację:

a) P(A, B, B), P(x ,y ,z).
b) Q(y, G(A, B)), Q(G(x, x), y).
c) Starszy(Ojciec(y), y)), Starszy(Ojciec(x), Jan).
d) Zna(Ojciec(y), y), Zna(x, x)

2. Zapisz logiczne reprezentacje następujących zdań w taki sposób, by były odpowiednie dla uogólnionej zasady Modus Ponens.

a) Konie, krowy i świnie są ssakami
b) Potomek konia jest koniem
c) *BiałyHuragan* jest koniem
d) *BiałyHuragan* jest ojcem *OgnistejKuli*
e) Potomek i rodzic są przeciwnymi relacjami
f) Każdy ssak ma rodzica

3. Z faktu „Konie są zwierzętami" wnioskujemy, że „Głowa konia jest głową zwierzęcia". Zademonstruj poprawność takiego wnioskowania poprzez następujące kroki:

a) Przetłumacz warunki i wniosek na zdania w logice pierwszego stopnia. Użyj trzech predykatów: GłowaOd(k,x), Koń(x) i Zwierze(x).
b) Zaneguj wniosek i przekształć warunki i zanegowany wniosek do postaci normalnej

4. Porównaj na dowolnym przykładzie wnioskowanie wprzód i w tył.

6 Planowanie

W tym rozdziale wprowadzimy podstawowe pojęcia i techniki stosowane przy konstrukcji agentów planowania, czyli systemów, które konstruują plan działania, na którym znajdują się kolejne cele. Agent planowania stara się osiągnąć te cele, by w ostateczności osiągnąć stan końcowy. W dalszej części tego rozdziału przedstawimy bardziej rozwinięte systemy planowania, rozwiązujące problemy, z którymi nie poradziłby sobie **prosty agent planowania**. Przedstawimy **algorytm planowania częściowego porządku**, który przeszukuje przestrzeń planów, by znaleźć ten, który na pewno zostanie osiągnięty. Zastosowanie takiego algorytmu pozwala nam na funkcjonowania naszych agentów w światach bardziej skomplikowanych.

6.1 Prosty agent planowania

Prosty agent planowania jest stosowany, gdy świat jest mały, a każdy jego stan jest dostępny tzn. agent może na podstawie swoich spostrzeżeń podać jednoznaczny stan świata. Wówczas, gdy wskażemy agentowi cel końcowy(ang. Goal), ten wywoła odpowiedni algorytm planowania (który nazwiemy Ideal-Planner) by wygenerować plan kolejnych akcji. Na podstawie tego planu agent będzie wykonywał znajdujące się w nim kolejne akcje.

W algorytmie, którym będzie się posługiwał Prosty agent planowania zastosujemy funkcje TELL i ASK odwołujące się bezpośrednio do bazy wiedzy KB. Funkcja Stwórz-Wyrażenie(*spostrzeżenie, t*) będzie odpowiadała za stworzenie w danym czasie *t* wyrażenia w języku bazy wiedzy na podstawie spostrzeżeń agenta planowania. Symbol statyczny *p* jest ciągiem zadań, które agent musi jeszcze wykonać, stąd początkowo *p* jest inicjowane na Plan_Zero. Działanie algorytmu dzieli się na 2 części. Pierwsza część jest realizowana, gdy planem akcji do wykonania jest Plan_Zero. Wówczas algorytm korzysta z funkcji generującej plan (Ideal-Planner). Podczas tego procesu nie jest wykonywana żadna akcja. Druga część algorytmu jest odpowiedzialna za wykonanie pierwszej akcji znajdującej się w *p*. Gdy ta akcja jest wybierana, z planu *p* jest usuwana pierwsza akcja.

Jak widać, algorytm tylko raz tworzy na podstawie globalnych spostrzeżeń plan działania, a potem konsekwentnie go realizuje aż do wyczerpania akcji na planie *p*.

Po każdym wyborze akcji (nawet wówczas, gdy *akcja* = brak_akcji) algorytm informuje bazę wiedzy o akcji, która zostanie wykonana w czasie t, wywołując funkcję TELL(*KB*, Stwórz-Wyrażenie(*akcja,t*)).

Poniżej przedstawiony jest algorytm, z którego korzysta Prosty agent planowania:

```
function PROSTY_AGENT_PLANOWANIA(spostrzeżenie) returns akcja
    static KB - baza wiedzy zawierająca opisy akcji
           p – plan zadań do wykonania (początkowo Plan_Zero)
           t – licznik czasu początkowo zainicjowany na 0
    zmienne lokalne: G – cel końcowy
                     Akt – aktualny opis stanu.

    TELL(KB, Stwórz-Wyrażenie(spostrzeżenie,t))
    Akt←OPIS_STANU(KB, t)
    if p=Plan_Zero then
           G←ASK(KB, Stwórz-Zapytanie(t))
           p ← Ideal-Planner(Akt, G, KB)
    if p = Plan_Zero or p = Plan_Pusty then
           akcja = brak_akcji
    else
           akcja ← PIERWSZA(p)
           p ← RESZTA(p)
    TELL(KB, Stwórz-Wyrażenie(akcja,t))
    t ← t+1
return akcja
```

6.2 Rozwiązywanie problemów i planowanie

Podamy teraz kilka istotnych informacji w jaki sposób należy konstruować plan na podstawie postawionego problemu. Zobaczmy też jakie są główne problemy, z którymi nie radzą sobie prości agenci planowania.

Każdy system planowania bazuje na 4 elementach:

- **akcje:** są one odpowiednikiem łączy pomiędzy stanami. Innymi słowy, przejście od stanu A do stanu B jest możliwe tylko wtedy, gdy istnieje jakaś akcja *a* łącząca te stany.
- **stany:** reprezentują one rzeczywistość widzianą przez agenta. Są one konsekwencją jego spostrzeżeń w kolejnych chwilach czasu. Stanem może być np. aktualny rozkład figur na szachownicy.
- **stany końcowe:** Prosty agent planowania widzi tylko stany początkowe i stany końcowe, a wszystko oprócz tego jest przez

niego uważane jako typowa „czarna skrzynka”. To właśnie przez to Prosty agent planowania nie widzi stanów wewnątrz tej skrzynki, które mogłyby być użyteczne w drodze do celu końcowego.

- **plany:** są reprezentowanie jako sekwencje akcji do wykonania. Podczas konstruowania rozwiązania plany biegną zawsze od stanów początkowych do stanów końcowych.

Mając tak zdefiniowany podział spróbujmy przeanalizować konstrukcję planu na poniższym przykładzie:

„Chcemy mieć sandały, ziemniaki i miód”. Zakładamy przy tym, że owych produktów nie posiadamy i chcemy je w jakiś sposób zdobyć. Niech w stanie początkowym agent znajduje się w domu. Akcje są wszystkim tym, co agent (czyli każdy człowiek) mógłby w danej chwili robić. Musimy jeszcze sprecyzować funkcję, która będzie szacowała odległość stanu aktualnego od stanu końcowego. Przyjmijmy za tą funkcję ilość przedmiotów (ze zbioru: {sandały, ziemniaki, miód}), które agent posiada w danej chwili.

Na rysunku nr 9 widzimy w jaki sposób gwałtownie rozbudowuje się przestrzeń poszukiwań. Mimo, iż istnieje jakieś rozwiązanie, to by do niego dotrzeć musimy rozpatrzyć ogromną ilość stanów nieużytecznych.

Rysunek 9 Rozwiązywanie problemu zakupów przy przeszukiwaniu wprzód przestrzeni wszystkich możliwych stanów

Z powyższego rysunku możemy od razu odczytać podstawową niedogodność. Algorytm planowania rozpatruje wszystkie możliwe stany, przez co dla dużych światów jest niewykonalny. Prawdziwym problemem jest też to, że funkcja szacująca jako argument bierze konkretny stan, stąd musi ona sprawdzić wszystkie stany, by wiedzieć, który jest najbliżej celu końcowego. Gdy agent znajdzie się już w supermarkecie i przykładowo będzie mógł wykonać akcje „Kup gruszki”, „Kup miód”, „Płać w kasie” to funkcja szacująca

da wynik „źle”, „dobrze”, „źle”. Agent wie, że kupienie miodu jest rzeczą dobrą ale nie ma żadnego pomysłu co zrobić dalej i musi myśleć od samego początku i rozpatrywać wszystkie dopuszczalne akcje. Informacja, że agent znajduje się w stanie początkowym w domu także nie upraszcza mu zadania. Może on bowiem pójść w wiele różnych miejsc, ponadto wcale nie wie w jaki sposób uzyskać pożądane przedmioty, czy je pożyczyć, kupić, ukraść?

Pierwszym sposobem planowania jest „otwarcie” reprezentacji stanów, akcji i celów końcowych. Algorytmy planowania bazują na logice pierwszego stopnia, dlatego możliwe jest rozbicie złożonych zdań na zdania krótsze, możliwe do analizowania. Ponadto jedne zdania mogą wynikać z innych i w ten sposób znacznie ograniczamy przestrzeń poszukiwań algorytmu: gdy wiemy że np.: następstwem Kup(x) jest Ma(x), wówczas gdy chcemy dojść do stanu Ma(miód) musimy najpierw znaleźć się w stanie Kup(miód).

Drugim sposobem planowania jest dodanie do planu zadań wtedy, gdy to jest potrzebne, a nie w sposób sekwencyjny jak to miało miejsce wcześniej. Agent może zdecydować, że kupi miód jeszcze zanim wyjdzie z domu, przez co będzie wiedział, że miód należy kupić (a nie np. ukraść) , a jego drzewo decyzyjne zdecydowanie się zmniejszy.

Trzecim sposobem polepszenia planowania jest przyporządkowywanie różnych stanów do różnych grup. To daje nam możliwość równoległego rozpatrywania pod-planów w poszczególnych grupach, które składają się z mniejszej ilość stanów. Ponadto, będąc w jednej grupie nie musimy rozpatrywać akcji, które są charakterystyczne dla innej grupy i co za tym idzie, drzewo decyzyjne znów staje się mniejsze. Stosujemy tu zasadę dziel i zwyciężaj, która jednak może okazać się nietrafna, bowiem zdarza się, że koszt szacowania ze sobą pod – planów jest zbyt duży. Grając w szachy nie możemy analizować osobno różnych figur lub różnych części szachownicy, bowiem może to doprowadzić do szybkiej przegranej.

6.3 Planowanie w rachunku sytuacyjnym.

Zanim zagłębimy się w techniki planowania, przeanalizujmy nasz problem „z zakupami” przy pomocy rachunku sytuacyjnego:

- **Stan początkowy**: Jeśli agent zaczyna działać w sytuacji S_0, wówczas stan początkowy może wyglądać następująco:

$W(dom,S_0) \wedge \neg Ma(sandały,S_0) \wedge \neg Ma(miód,S_0) \wedge \neg Ma(ziemniaki,S_0)$

- **Stan końcowy**: jest osiągnięty, gdy spełnione jest zdanie:

$\exists s \; W(dom,s) \wedge Ma(sandały,s) \wedge Ma(miód,s) \wedge Ma(ziemniaki,s)$

- **Operatory:** Zbiór opisów akcji, np. aksjomat kontynuacji w czasie dla akcji Kup(miód) :

$$\forall a,s \;\; Ma(miód,Rezultat(a,s)) \Leftrightarrow [(a=Kup(miód) \wedge W(Supermarket,s)) \vee (Ma(miód,s) \wedge a \neq Upuść(miód))]$$

Dla przypomnienia, funkcja Rezultat(a,s) zwraca stan, który jest wynikiem działania akcji *a* w stanie *s*. Funkcję tą rozszerzymy do funkcji Rezultat'(l,s), która będzie zwracała stan będący wynikiem działania sekwencji akcji *l*, począwszy od stany *s*. Będziemy mieli zatem:

$\forall s \;\; Rezultat'([],s) = s$

$\forall a,l,s \;\; Rezultat'([a|l], s) = Rezultat'(l ,Rezultat(a,s))$

Rozwiązaniem naszego problemu jest więc dla stanu początkowego S_0 taki plan p, który spełnia poniższe zdanie:

$W(dom, Rezultat'(p,S_0)) \wedge Ma(sandały, Rezultat'(p,S_0)) \wedge Ma(miód, Rezultat'(p,S_0)) \wedge Ma(ziemniaki, Rezultat'(p,S_0))$

Poprawny plan *p* mógłby zatem wyglądać następująco:

p=[Idź(Supermarket), Kup(miód), Kup(ziemniaki), Idź(SklepObuwniczy), Kup(sandały), Idź(Dom)].

6.4 Podstawowa reprezentacja planowania.

Reprezentacja stanów

W języku STRIPS, wszystkie stany (W tym również początkowe i końcowe) są reprezentowane przez koniunkcję odpowiednich wyrażeń, którymi są predykaty zastosowane do symboli stałych. Dla przykładu, początkowy stan dla problemu „z zakupami" będzie wyglądał następująco:

$W(dom,S_0) \wedge \neg Ma(sandały,S_0) \wedge \neg Ma(miód,S_0) \wedge \neg Ma(ziemniaki,S_0)$

W wielu systemach zdarza się, że cel końcowy nie jest kompletny. Sytuacja taka może wystąpić w przypadku działania w nie do końca dostępnym środowisku. W takiej sytuacji agent sam rozszerza na wszystkie sposoby swój plan niekompletny, tworząc różne wersje planu kompletnego.

Niektóre systemy wprowadzają jednak konwencję: „błędem jest negacja", która mówi, że jeśli w opisie stanu nie znajduje się dane pozytywne wyrażenie, to wyrażenie to jest uznawane jako fałszywe.

Stany końcowe mogą również zawierać zmienne, np. jeśli stanem końcowym jest przebywanie w sklepie, który sprzedaje obuwie, wówczas będzie mu odpowiadało zdanie:

$W(x) \wedge Sprzedaje(x, Obuwie)$

Reprezentacje akcji

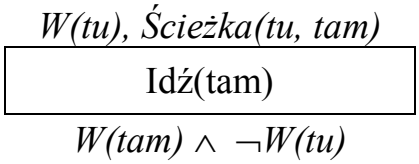
Operatory języka STRIPS składają się z 3 komponentów:

- **Opis akcji**: jest tym, co agent w rzeczywistości przekazuje do środowiska, w którym działa. Dla projektanta planu, jest to tylko nazwa możliwej akcji.
- **Warunek**: jest koniunkcją atomów (pozytywnych wyrażeń), które mówią o tym co musi być prawdziwe zanim akcja będzie możliwa do wykonania.
- **Efekt** akcji jest koniunkcją wyrażeń (prawdziwych lub fałszywych), które opisują w jaki sposób zmieni się świat po wykonaniu przez agenta danej akcji.

Poniżej podany jest przykład składni STRIPS użytej do wyrażenia faktu, że agent może się przemieszczać z jednego miejsca do drugiego:

Op(AKCJA: *Idź(tam)*, WARUNEK: $W(tu) \wedge Ścieżka(tu,tam)$,
EFEKT: $W(tam) \wedge \neg W(tu)$)

Będziemy także posługiwali się graficznymi oznaczeniami operatorów, co dla powyższego przykładu będzie wyglądało następująco:



Rysunek 10 Diagram dla operatora Idź(tam). Warunki są powyżej akcji, a efekty poniżej

Mówimy, że operator o jest odpowiedni w stanie s, jeśli wszystkie warunki operatora o są spełnione w stanie s, tzn. Warunki(o) $\subset$ s. W nowym stanie (będącym rezultatem działania operatora) spełnione są wyrażenie dane wzorem: $NoweWyrażenia = [S + E^{+}(o)] - E^{-}(o)$, gdzie S jest zbiorem wyrażeń, które były spełnione przed działaniem operatora o, $E^{+}(o)$ jest zbiorem wyrażeń pozytywnych będących efektem operatora o, natomiast $E^{-}(o)$ jest zbiorem wyrażeń negatywnych będących efektem operatora o. Dla przykładu, jeśli stan początkowy zawiera wyrażenia:

W(dom) , Ścieżka(dom, supermarket), ...

wówczas akcja Idź(Supermarket jest dopuszczalna), a stan końcowy zawiera wyrażenia:

$\neg$W(dom), W(supermarket), Ścieżka(dom, supermarket), ...

Przestrzeń sytuacji i przestrzeń planu

Wiemy, że każdy plan jest odpowiednim grafem skierowanym, który posiada źródło (wierzchołek z którego rozpoczynamy wykonywanie akcji) i wierzchołki końcowe, które utożsamiamy ze stanami końcowymi.

Pierwszym sposobem konstrukcji planów jest rozpatrywanie **przestrzeni sytuacji**. Sposób ten dzieli się na 2 kategorie. Pierwszą kategorią jest rozpatrywanie wszystkich możliwych sytuacji począwszy od sytuacji początkowej (startowej). Tak jak to miało miejsce na rysunku nr 1. przeszukujemy po kolei wszystkie możliwe odgałęzienia w nadziei, że znajdziemy ścieżkę prowadzącą do celu. Taka konstrukcja planu oparta jest na **progresywnym** przeszukiwaniu przestrzeni sytuacji, a co za tym idzie, posiada główną wadę: konieczność przeszukiwania wszystkich możliwych ścieżek w grafie. Algorytm wykorzystujący tą metodę jest zatem wolny i potrzebuje do działania ogromnych rozmiarów pamięci.

Dobrym sposobem na obcięcie wielu ścieżek poszukiwań jest zastosowanie algorytmu **regresywnego**, będącego odpowiednikiem przeszukiwania w tył. Taki algorytm jest możliwy do zastosowania, ponieważ każdy operator posiada listę wyrażeń, które są warunkami jego wykonania. Możemy więc wystartować od stanu końcowego, którym jest lista wyrażeń, a następnie sprawdzić jakie akcje dają w rezultacie wyrażenia z tej listy. Teraz z kolei nowe akcje posiadają nowe warunki, które tworzą nowe stany, do których chcemy dojść, i algorytm rekursywnie kontynuuje działanie. Jeśli jakaś ścieżka nie doprowadzi nas do stanu początkowego, wówczas cofamy się do wcześniejszego stanu i wykonujemy kolejną akcję, która jest odpowiednia w tym stanie itd. Niestety, przeszukiwanie w tył staje się skomplikowane i niekiedy niewykonalne, gdy mamy do czynienia z planem, który posiada kilka stanów końcowych.

Podsumowując algorytmy bazujące na przestrzeni sytuacji stwierdzamy, że w przypadku algorytmów progresywnych budują one drzewo planu od korzenia (stanu początkowego) do liścia (stanu końcowego), lub w przypadku algorytmów regresywnych od liścia do korzenia. W każdym etapie działania tych algorytmów do nowo tworzonego drzewa dodawany jest nowy wierzchołek reprezentujący nową sytuację.

Alternatywnym sposobem działania są algorytmy rozpatrujące **przestrzeń planów**. Startują one od jak najmniejszych planów częściowych, a następnie je rozszerzają by w rezultacie osiągnąć plan końcowy rozwiązujący problem. Operatorami w tym przeszukiwaniu są operatory planu: dodanie stanu, wprowadzenie porządku (jeden stan przed drugim), powiązanie nowej zmiennej itp. Rezultatem jest końcowy plan, a ścieżka prowadząca do niego nie jest istotna.

Algorytmy rozpatrujące przestrzeń planów dzielą się na dwa typy: jedne rozpatrują wszystkie plany i nakładają na nie ograniczenia w wyniku czego nigdy nie dodają nowych planów, lecz eliminują te, która nie pasują do nowych ograniczeń. Drugim typem jest konstruowanie planów, a następnie ich modyfikowanie, by pasowały do ograniczeń.

Reprezentacje dla planów

Zajmiemy się teraz rozpatrywaniem przestrzeni planów. W tym celu będziemy musieli wprowadzić notację pozwalającą na analizowanie kolejnych planów. Dla przykładu przeanalizujemy problem zakładania przedniego i tylniego koła na rower. Celem będzie złożenie dwóch wyrażeń: PrzednieKołoZał $\wedge$ TylnekołoZał, w stanie początkowym nie ma żadnych wyrażeń, a czterema operatorami są:

Op(AKCJA: *PrzednieKoło,* WARUNEK: *PrzedniaDętkaZał*,
EFEKT:*PrzednieKołoZał*)
Op(AKCJA: *TylneKoło,* WARUNEK: *TylnaDętkaZał*,
EFEKT: *TylneKołoZał*)
Op(AKCJA: *PrzedniaDętka,* EFEKT: *PrzedniaDętkaZał*)
Op(AKCJA: *TylnaDętka,* EFEKT: *TylnaDętkaZał*)

Częściowy plan dla tego problemu zawiera dwa kroki: *PrzednieKoło* i *TylneKoło*, jednak nie wiemy przecież który z tych kroków powinien zostać wykonany jako pierwszy. Spotykamy się tutaj z planem częściowym, dla którego powyższa kolejność nie jest istotna (może ona zostać ustalona w późniejszym etapie planowania). To jest stosunkowo dobre podejście przy konstruowaniu planów, ponieważ często zdarza się, że podejmujemy nieistotne decyzje, które w późniejszym etapie planowania mogą prowadzić do kolizji. Gdy natomiast dodajemy do planu nowy krok PrzedniaDętka, wówczas żądamy, by algorytm umieścił ten krok w odpowiedniej kolejności (tzn. dla powyższego przykładu przed operacją Przednie koło). Alternatywą jest plan totalny, który składa się ze zwykłej listy kolejnych kroków do wykonania. Przekształcanie planu częściowego P do planu totalnego P nazywa się **linearyzacją planu** P.

Przykład z kołami rowerowymi nie pokazuje jednak ważnego elementu, który używamy przy konstrukcji planów: **powiązań**. Załóżmy, że naszym celem jest *Ma(mleko)* i dysponujemy akcją *Kup(produkt, sklep)*. Intuicja podpowiada nam, by związać zmienną „*produkt*” ze stałą „*mleko*”. Okazuje się jednak, że powiązanie zmiennej „*sklep*” z określonym sklepem nie jest dobrym pomysłem, bowiem może się okazać, że innym celem będzie zakup w innym sklepie, a zmienna „*sklep*” będzie już posiadała kolidujące przypisanie. Warto więc przypisania, których nie jesteśmy pewni zostawić na później. Ta strategia pozwala nam również na pozbycie się złych planów. Załóżmy przykładowo, że gałąź drzewa poszukiwań zawierającego wyrażenie Kup(Mleko, sklep)

prowadzi nas do kolizji nie związanej z wyborem sklepu (np. brak pieniędzy w portfelu, lub zepsuta karta kredytowa). Jeśli zmienna „*sklep*" została już przypisana do określonego sklepu, to algorytm cofnie się i zamieni powiązanie zmiennej „*sklep*" na inne miejsce, które z powodu braku funduszów i tak nie doprowadzi nas do pozytywnego rozwiązania. Jeśli natomiast brak jest powiązania „*sklepu*" z konkretnym miejscem, wtedy algorytm cofnie się dalej (nie będzie rozpatrywał innych sklepów) i być może natrafi na gałąź prowadzącą do celu (np. zawierającą krok „*PożyczPieniądze*"). Plany, w których każda zmienna jest związana nazywamy Planami **całkowicie zainicjowanymi**.

Będziemy odtąd używali takiej reprezentacji planów, która pozwoli nam na późniejsze dookreślanie porządku oraz powiązań zmiennych. **Plan** jest formalnie zdefiniowany jako struktura danych składająca się z czterech komponentów:

- Zbiór kroków na planie. Każdy krok jest jednym z operatorów występujących w problemie
- Zbiór przypisań porządku. Każde przypisanie ma postać $S_i \langle S_j$ co oznacza, że krok S_i występuje jakiś czas przed krokiem S_j (ale nie koniecznie bezpośrednio przed)
- Zbiór powiązań zmiennych. Każde powiązanie ma postać $v = x$, gdzie v jest zmienną, którą wiążemy, a x jest stałą lub inną zmienną
- Zbiór **połączeń przyczynowych**. Połączenie przyczynowe ma postać $S_i \xrightarrow{C} S_j$ i służy do zapamiętywania celów kroków na planie. W tym przypadku celem S_i jest osiągnąć warunek c dla stanu S_j.

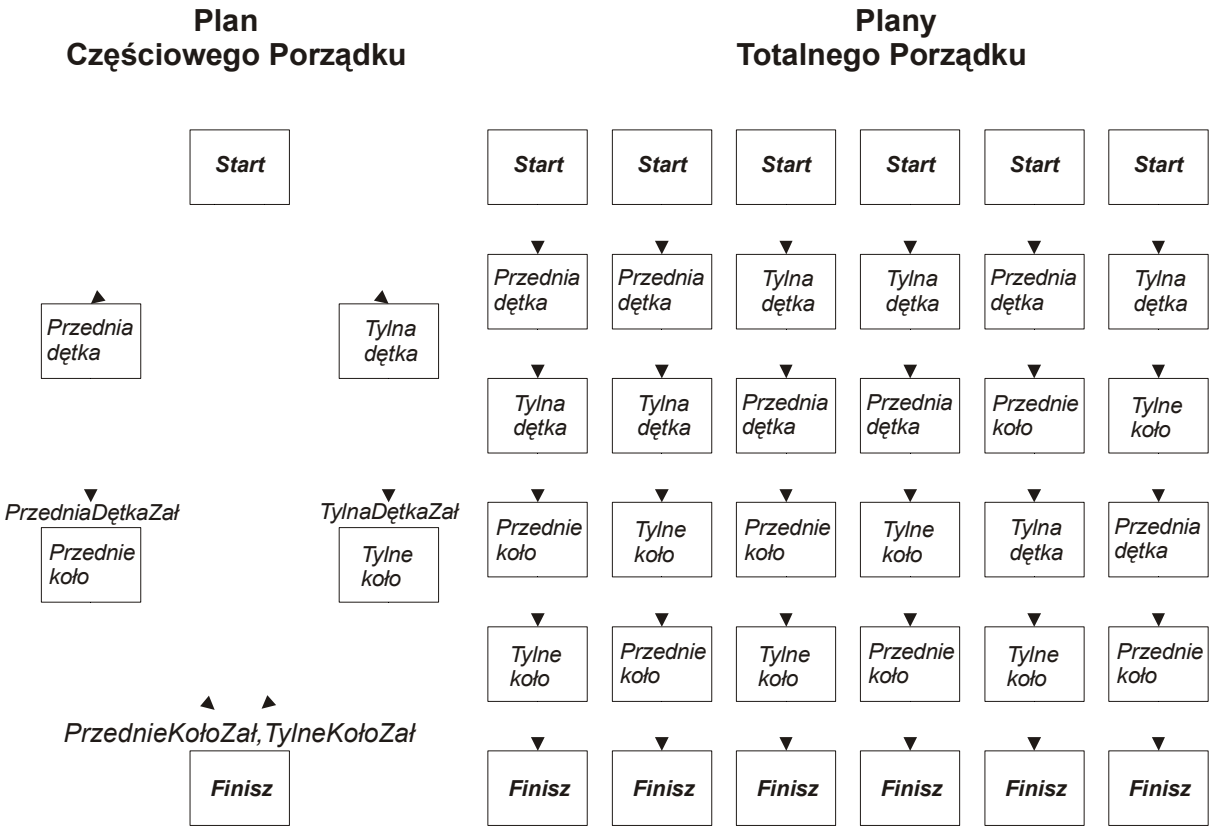
Stan inicjujący, zanim nałożymy na niego jakiekolwiek ograniczenia opisuje po prostu nierozwiązany problem. Składa się z dwóch kroków : Start i Finisz ustawionych w porządek Start $\langle$ Finisz. Z krokami tymi nie są związane żadne akcje, więc gdy wykonujemy plan, są one ignorowane. Start nie posiada żadnych warunków, a jako efekt posiada wszystkie wyrażenia, które były prawdziwe na początku konstruowania planu. Finisz posiada jako warunek listę wyrażeń, która jest celem planu; Finisz jest ostatnim krokiem i nie posiada efektów. W celu rozwiązania problemu startuje się z planu inicjującego, a następnie stopniowo się go rozszerza, by dojść do rozwiązania. Stan inicjujący dla problemu „zakładania kół na rower" jest przedstawiony na rysunku nr 11.

Rysunek 11 Plany inicjujące

Zapis stanu inicjującego wygląda następująco:

Plan(KROKI: { S_1:Op(AKCJA: *Start*),
S_2:Op(AKCJA: *Finisz*,
WARUNEK: *PrzednieKołoZał* $\wedge$ *TylneKołoZał*)},
PORZĄDKI: {$S_1 \langle S_2$},
POWIĄZANIA: {},
POŁĄCZENIA: {})

Na rysunku nr 12 pokazany jest plan częściowy rozwiązujący ten problem i 6 możliwości linearyzacji tego problemu. Widać, że plany częściowe są bardzo wygodnym sposobem przedstawiania rozwiązań, które nie uwzględniają wyborów nie mających wpływu na poprawność planu. Wraz z liniowym wzrostem ilość kroków planu, ilość linearyzacji wzrasta ekspotencjalnie. Przykładowo gdybyśmy do problemu dodali zakładanie na rower siodełka i dzwonka, (dwóch kroków nie mających wpływu na inne kroki) wówczas istniałoby 180 linearyzacji jednego planu częściowego.



Rysunek 12 Plan częściowy dla "Problemu zakładania kół na rower" i jego 6 linearyzacji

Rozwiązania

Rozwiązaniem jest plan, który agent może wykonać i który gwarantuje osiągnięcie celu końcowego. Jeśli chcielibyśmy łatwo sprawdzić, czy plan jest rozwiązaniem, wówczas moglibyśmy uważać za rozwiązania tylko plany totalne. Zawężanie się tylko do planów totalnych nie jest dobrym pomysłem z 3 powodów: Po pierwsze plany częściowe są czytelniejsze niż ich linearyzacje. Po drugie, niektóre plany częściowe mogą być wykonywane równolegle, dlatego podział czynności na różne jednostki obliczeniowe przyspiesza wykonywanie planu. Linearyzacja narzuca sekwencyjne wykonywanie wszystkich kroków. Po trzecie, gdy tworzymy plany częściowe, które będziemy później składać z innymi planami, podział zadań na osobne, równolegle wykonywane grupy znacznie zwiększa elastyczność planu końcowego.

Dopuszczamy więc rozwiązania, którymi są plany częściowe używając prostej definicji: Rozwiązaniem jest taki plan częściowy, który jest **kompletny i konsekwentny**.

Plan jest **kompletny**, jeśli każdy warunek każdego kroku jest osiągalny przez jakiś inny krok. Krok osiąga warunek, jeśli warunek jest jednym z efektów tego kroku i jeśli żaden inny krok nie może wyeliminować tego warunku. W zapisie formalnym: krok S_i osiąga warunek c kroku S_j, jeśli (1) $S_i \langle S_j$ oraz $c \in EFEKT(S_i)$ i (2) nie istnieje krok S_k taki, że $(\neg c) \in EFEKT(S_k)$ oraz w jakiejś linearyzacji planu występuje $S_i \langle S_k \langle S_j$.

Plan jest **konsekwentny** gdy w zbiorze przypisań porządku i w zbiorze połączeń nie ma żadnych sprzeczności. Sprzeczność występuje, gdy występuje zarówno $S_i \langle S_j$ jak i $S_j \langle S_i$ lub gdy występuje v=A i v=B (dla różnych wartości stałych A i B). Obie relacje: = i $\langle$ są tranzytywne, tzn. plan zawierający $S_i \langle S_j$, $S_j \langle S_k$ oraz $S_k \langle S_i$ jest niekonsekwentny.

Plan częściowy z rysunku nr 12 jest rozwiązaniem, ponieważ wszystkie warunki są osiągnięte. Widzimy także że każda jego linearyzacja jest także rozwiązaniem, stąd plan jest kompletny i konsekwentny.

6.5 Przykład tworzenia planu częściowego porządku.

W tej sekcji prześledzimy działanie algorytmu budującego plan częściowego porządku. Algorytm zaczyna funkcjonować mając tylko dwa kroki: Start i Finisz i w każdej iteracji dodaje jeden krok. Jeśli prowadzi to do planu niekonsekwentnego, cofa się i próbuje wybrać inną gałąź w przestrzeni poszukiwań. By móc skupić się na przeszukiwaniu, będziemy rozpatrywać dodawanie tylko takich kroków, które pomogą osiągnąć nieosiągnięte dotychczas warunki. Połączenia przyczynowe będą służyły do zapamiętania przebiegu tego procesu.

Algorytm zastosujemy do problemu, o którym była mowa na początku tego rozdziału: chcemy, by agent, który znajduje się w domu kupił ziemniaki, miód i sandały i by wrócił do domu. Wprowadzimy pewne założenia: akcja „Idź" będzie pozwalała agentowi na przemieszczanie się pomiędzy dowolnymi lokalizacjami; opis akcji „Kup" nie będzie uwzględniał ilości pieniędzy w portfelu agenta. Dla przejrzystości zapisu napiszemy SM zamiast Supermarketu oraz SO zamiast Sklepu obuwniczego.

Przedstawimy proces konstrukcji planu, który doprowadzi do końcowego rozwiązania. Sam kod źródłowy algorytmu podany zostanie w dalszej części tego rozdziału.

Stan inicjacyjny jest zdefiniowany przez poniższy operator:

Op(AKCJA:*Start*, EFEKT:*W(dom)* $\wedge$ *Sprzedaje(SO,sandały)* $\wedge$
Sprzedaje(SM,miód) $\wedge$ *Sprzedaje(SO,ziemniaki)*)

Stan końcowy (cel) jest zdefiniowany przez krok Finisz, precyzujący, które obiekty agent musi posiadać i gdzie musi się znajdować:

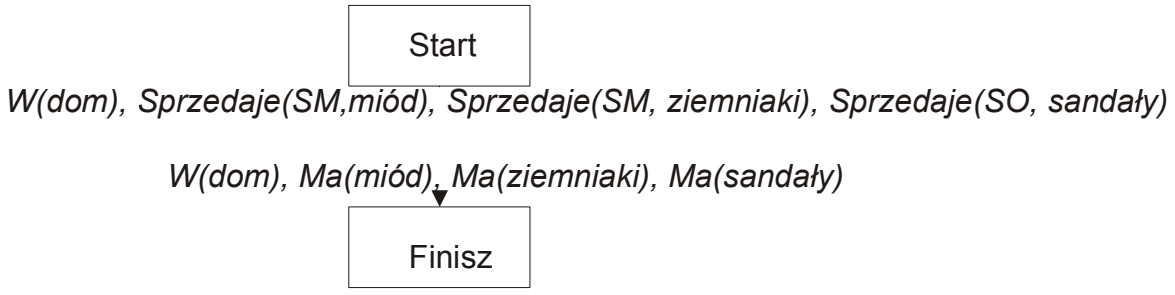
Op(AKCJA:*Finisz*, WARUNEK:*W(dom)* $\wedge$ *Ma(sandały)* $\wedge$
Ma(miód) $\wedge$ *Ma(ziemniaki*))

Akcje są zdefiniowane w sposób następujący:

Op(AKCJA: *Idź(tam)*, WARUNEK: *W(tu)*
EFEKT: *W(tam)* $\wedge$ $\neg$*W(tu)*)

Op(AKCJA: *Kup(x)*, WARUNEK: *W(sklep)* $\wedge$ *Sprzedaje(sklep,x)*
EFEKT: *Ma(x)*)

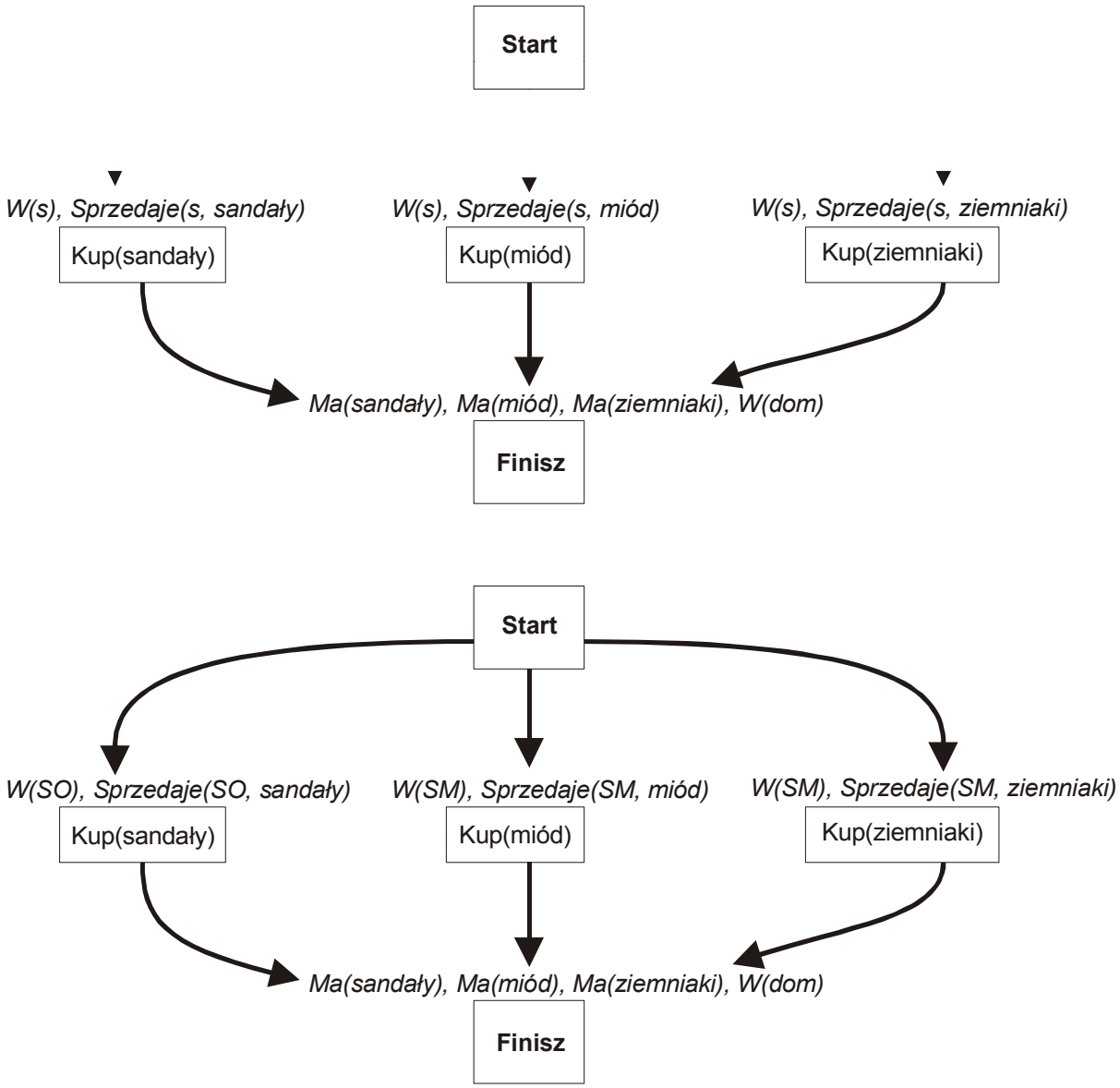
Diagram dla planu inicjacyjnego jest następujący:



Rysunek 13 Diagram planu inicjacyjnego dla problemu "zakupów"

Już na początku widzimy, że jest wiele dróg, według których plan inicjacyjny może być rozwinięty. Niektóre wybory będą działały, inne nie. Dla ilustracji będziemy najpierw wybierali dobre akcje, a później kilka złych. Na rysunku nr 14 przedstawiony jest wybór trzech akcji Kup, które pozwalają na osiągnięcie warunków kroku Finisz. Nie mamy innego wyboru, ponieważ w bibliotece akcji tylko akcja Kup daje nam własność posiadania jakiegoś produktu.

Pogrubione strzałki na rysunku nr 14 symbolizują połączenia przyczynowe. Dla przykładu: Znajdujące się najbardziej na prawo połączenie przyczynowe oznacza, że krok *Kup(sandały)* został dodany w celu osiągnięcia warunku kroku *Finisz Ma(Sandały)*. Algorytm musi od teraz zabezpieczać ten warunek: jeśli jakiś krok miałby zmazać warunek *Ma(Sandały)* to nie zostanie on wstawiony pomiędzy krok *Kup(Sandały) i Finisz*. Cienkie strzałki reprezentują relację porządku. Z definicji wszystkie akcje muszą się wywodzić z kroku *Start*. Ponadto wszystkie akcje wymuszają efekty, które są ich konsekwencjami, dlatego każda gruba strzałka jest zarazem strzałką cienką.

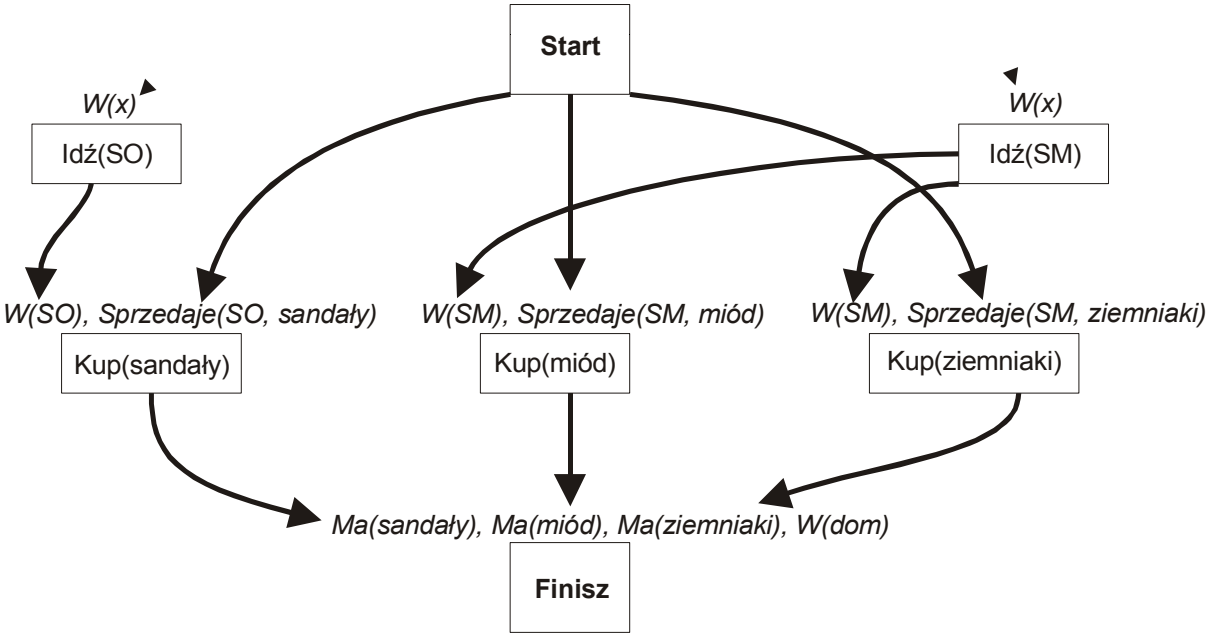


Rysunek 14 Problem "Zakupów" - 1 etap działania algorytmu

Drugi etap na rysunku nr 14 pokazuje sytuację, gdy algorytm zdecydował się osiągnąć warunki Sprzedaje łącząc je ze stanem inicjacyjnym. Znów, algorytm nie ma innego wyboru, ponieważ nie ma żadnego innego operatora, który osiąga *Sprzedaje*.

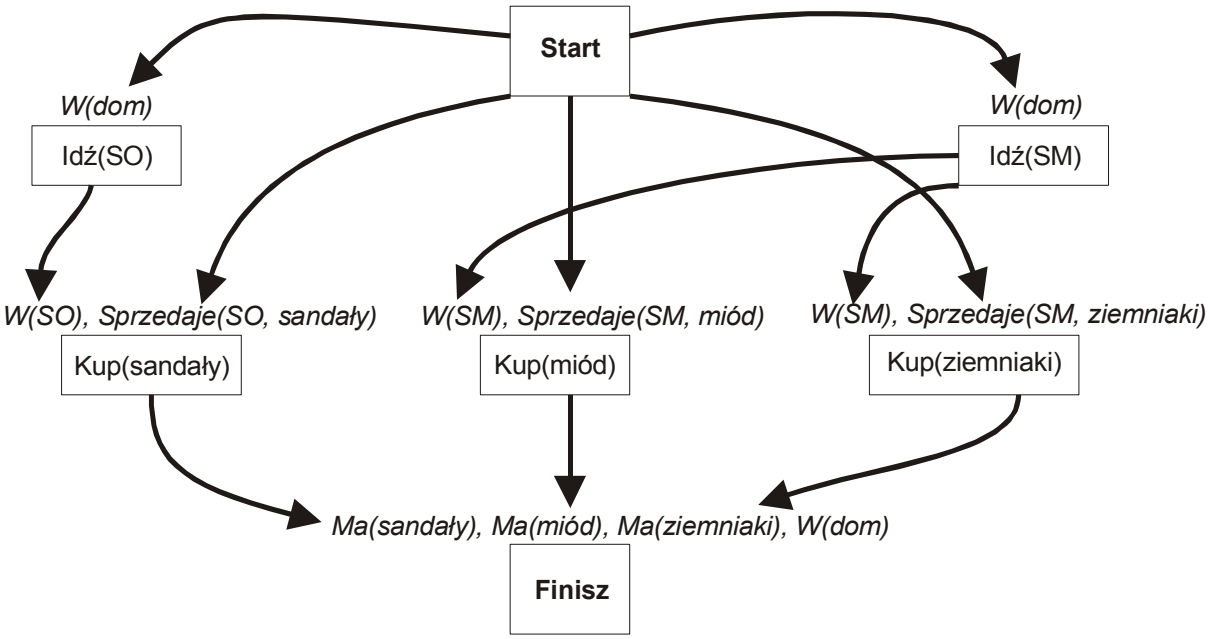
Mimo, iż może się wydawać, że zrobiliśmy niewiele, jest to znaczące ulepszenie w stosunku do rozwiązania problemu metodą przeszukiwania wprzód. Po pierwsze, ze wszystkich możliwych do kupienia rzeczy i wszystkich miejsc, do których można pójść wybraliśmy właściwe akcje *Kup* i właściwe miejsca, bez straty czasu na inne. Po wybraniu akcji nie musimy od razu ustawiać ich w określony porządek; algorytm tworzący plan częściowego porządku może podjąć tą decyzję później.

Na rysunku nr 15 rozszerzamy nasz plan wybierając dwie akcje *Idź* (prowadzące odpowiednio do Supermarketu i Sklepu Obuwniczego) w celu osiągnięcia warunków „*W*" dla kolejnych akcji *Kup*.



Rysunek 15 Problem "Zakupów" - 2 etap działania algorytmu

Aż do tego czasu wszystko przebiegało łatwo. Algorytm zaszedł daleko bez konieczności przeszukiwań. Teraz spotykamy się z pewnym problemem. Dwie akcje *Idź* posiadają nieosiągalne warunki, które na siebie wzajemnie oddziałują, ponieważ agent nie może być w tym samym czasie w dwóch różnych miejscach. Każda akcja *Idź* posiada warunek *W(x)*, gdzie x jest lokalizacją agenta przed wywołaniem akcji *Idź*. Załóżmy, że algorytm próbuje osiągnąć warunki *Idź(SO)* i *Idź(SM)* łącząc je z wyrażeniem *W(dom)* w stanie *Start*. Rezultatem jest plan pokazany na rysunku nr 16.

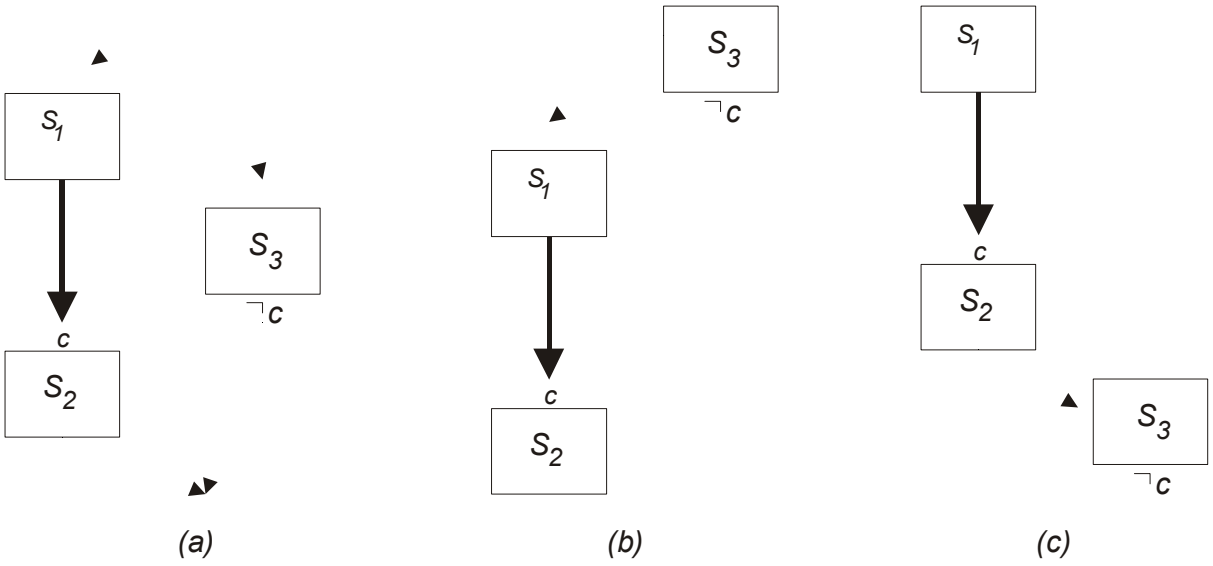


Rysunek 16 Problem "Zakupów" - 3 etap działania algorytmu

Niestety, to prowadzi nas do problemu. Krok *Idź(SO)* dodaje wyrażenie *W(SO)* ale kasuje wyrażenie W(dom). Jeśli więc agent pójdzie do sklepu obuwniczego, wówczas nie może już pójść z domu do supermarketu (tzn. dopóki nie wprowadzi dodatkowego kroku przejścia ze sklepu obuwniczego do domu – lecz to połączenie przyczynowe oznacza, że właśnie krok Start osiąga wyrażenie W(dom)). Z drugiej strony, jeśli agent pójdzie najpierw do supermarketu, wówczas nie może już pójść z domu do sklepu obuwniczego.

W tym momencie doszliśmy do ślepej uliczki w poszukiwaniu naszego planu, dlatego algorytm musi się cofnąć i wybrać inną drogę. Warto zastanowić się jak bez dużej straty czasu algorytm wykrywa, że dany plan częściowy prowadzi do ślepej uliczki. Kluczowe jest tutaj spostrzeżenie, że połączenia przyczynowe w planie częściowym są **połączeniami zabezpieczonymi**.

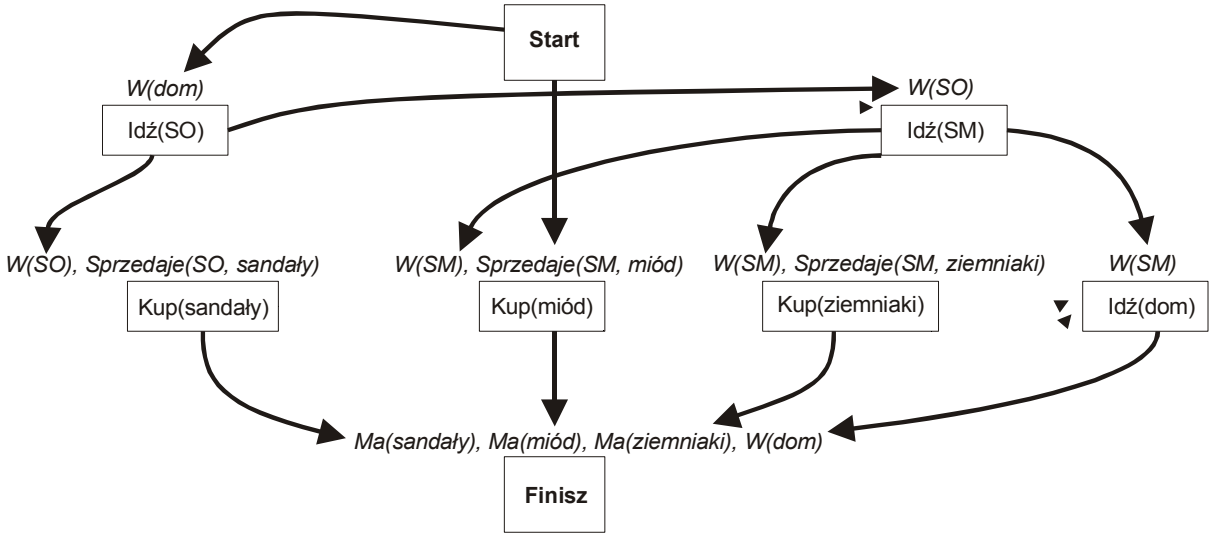
Połączenie przyczynowe jest zabezpieczane przez upewnienie się, że wszystkie **zagrożenia** (kroki, które mogą wymazać zabezpieczony warunek) są tak uporządkowane, że występują przed lub po zabezpieczonym połączeniu. Rysunek nr 17(a) pokazuje zagrożenie S_3 połączenia przyczynowego $S_1 \xrightarrow{C} S_2$ ponieważ jeden z efektów S_3 wymazuje „*c*”. Sposobem na rozwiązanie tego problemu jest dodanie odpowiednich porządków, by S_3 nie występował pomiędzy S_1 i S_2. Jeśli S_3 jest ustawiane przed S_1, mówimy o degradacji (Rysunek nr 17(b)), natomiast jeśli S_3 jest po S_2 to mówimy o promocji (Rysunek nr 17(c)).



Rysunek 17 Zabezpieczające połączenia przyczynowe

Na rysunku nr 18 nie ma możliwości rozwiązania zagrożenia, które każdy krok *Idź* przedstawia dla innych kroków *Idź*. Którykolwiek *Idź* przyjdzie pierwszy, usunie on wyrażenie *W(dom)* dla innego kroku. Za każdym razem, kiedy nie można przez promocję i degradację rozwiązać zagrożeń, algorytm cofa się i szuka innych dróg we wcześniejszym momencie procesu planowania.

Spróbujmy teraz inną drogą osiągnąć warunek *W(x)* dla kroku *Idź(SM)*, tym razem przez dodanie połączenia przyczynowego od *Idź(SO)* do *Idź(SM)*. Innymi słowy, plan jest taki, by pójść z domu do sklepu obuwniczego, a potem do supermarketu (Sytuacja ta jest pokazana na rysunku nr 18).



Rysunek 18 Problem "Zakupów" - 4 etap działania algorytmu

To pociąga za sobą następne zagrożenie. Dopóki plan nie jest bardziej uściślony, będzie on pozwalał agentowi na pójście ze sklepu obuwniczego do supermarketu bez wcześniejszego zakupu sandałów (czyli akcji, dla której agent

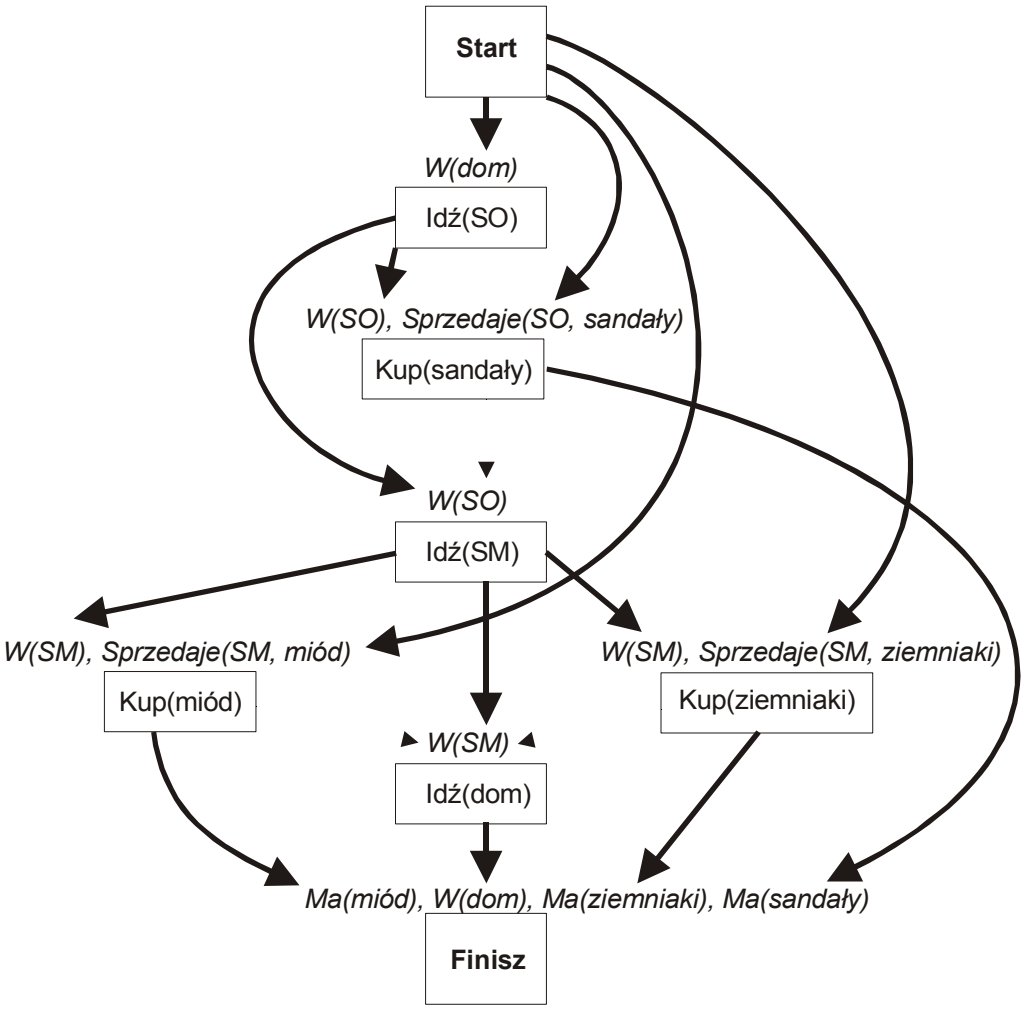
przyszedł do sklepu obuwniczego). Krok *Idź(SM)* zagraża warunkowi *W(SO)* dla kroku *Kup(sandały)*, który jest chroniony przez połączenie przyczynowe. Zagrożenie jest eliminowane przez wymuszenie na *Idź(SM)* by następowało po *Kup(sandały)*.

Tylko warunek *W(dom)* dla kroku *Finisz* pozostaje nieosiągnięty. Dodanie kroku *Idź(dom)* osiąga ten warunek, ale wprowadza inny warunek *W(x)*, który musi zostać osiągnięty. Znów, zabezpieczenie połączenia przyczynowego podpowie algorytmowi jak ma się zachować:

- Jeśli spróbuje osiągnąć *W(x)* przez połączenie do *W(dom)* w stanie *Start*, nie będzie sposobu, by usunąć zagrożenie spowodowane przez *Idź(SO)* i *Idź(SM)*.
- Jeśli spróbuje połączyć *W(x)* z krokiem *Idź(SO)*, nie będzie możliwe usunięcie zagrożenia ze strony kroku *Idź(SM),* na którym wymusiliśmy, by następował po *Idź(SO)*.
- Połączenie od *Idź(SM)* do *W(x)* oznacza, że x jest powiązany z *SM*, stąd krok *Idź(dom)* wymazuje warunek *W(SM)*.To powoduje zagrożenia warunków *W(SM)* dla kroków *Kup(miód)* i *Kup(ziemniaki)*. Zagrożenia te mogą jednak być wyeliminowane dodając porządek: *Idź(dom)* następuje do krokach *Kup(miód)* i *Kup(ziemniaki)*.

Rysunek nr 19 pokazuje kompletny plan rozwiązania, wraz ze wszystkimi zależnościami porządkowymi. Rezultatem jest prawie plan totalnie uporządkowany; jedyną rzeczą nieścisłą jest dowolna kolejność kroków *Kup(miód)* i *Kup(ziemniaki)*.

Możemy podsumować to, co omawiany algorytm tworzący plan częściowego porządku zdołał osiągnąć. Główną jego zaletą jest to, iż potrafi on rozwiązywać problemy oparte na dużej liczbie stanów. Ponadto jest on zorientowany na warunki poszczególnych kroków, przez co nie rozpatruje całego drzewa poszukiwań. Ostatecznie, połączenia przyczynowe informują algorytm kiedy należy porzucić aktualny plan bez straty czasu na rozbudowywanie kolidujących ze sobą części planu.



Rysunek 19 Rozwiązanie problemu z "Zakupami"

6.6 Algorytm tworzący plan częściowego porządku

W tej sekcji zapiszemy formalnie algorytm tworzący plan częściowego porządku, który analizowany był wcześniej na przykładzie problemu z „zakupami". Nazwiemy go POP (ang. Partial-Order-Planner) i będzie to algorytm niedeterministyczny, gdyż użyjemy w nim instrukcji „Wybierz" i wywołania „fail".

POP rozpoczyna pracę na minimalnym planie częściowym i w kolejnych etapach swojego działania rozszerza aktualny plan próbując osiągnąć warunek c pewnego kroku S_{akt}. Algorytm czyni to przez wybór pewnego operatora – albo spośród kroków już znajdujących się na planie, albo spośród całej puli operatorów – osiągającego warunek c. W dalszej części zapamiętuje połączenie przyczynowe dla nowo osiągniętego warunku, a następnie eliminuje wszystkie zagrożenia dla połączeń przyczynowych. Nowy krok może zagrażać istniejącemu połączeniu przyczynowemu, lub istniejący krok może zagrażać

nowemu połączeniu przyczynowemu. Jeśli w tym momencie algorytm nie zdoła wyeliminować zagrożenia, wówczas cofa się do wcześniejszego momentu tworzenia planu i szuka innego operatora. Warto zauważyć pewną subtelność algorytmu – wybór kroku, który musimy osiągnąć jak i jego warunku nie ulega zmianie przy cofaniu się algorytmu. Rozpatrując najpierw krok c_1, a potem c_2 otrzymamy ten sam plan, co rozpatrując najpierw c_2, a potem c_1. Możemy więc bez zastanawiania wziąć warunek do osiągnięcia bez obawy o back-tracking, nasz wybór nie ma wpływu na znajdywane rozwiązanie – jedynie na szybkość jego znalezienia.

function POP(*stan_początkowy, cel, Operatory*) **returns** *plan*

plan←STWÓRZ_MIN_PLAN(*stan_początkowy,cel*)
loop do
  if ROZWIĄZANIE?(*plan*) **then return** *plan*
  Sakt,c←WYBIERZ_PODCEL(*plan*)
  WYBIERZ_OPERATOR(*plan,Operatory,*S_{akt}*,c*)
  ELIM_ZAGROŻENIA(*plan*)
end

function WYBIERZ_PODCEL(*plan*) **returns** S_{akt}, *c*

wybierz krok S_{akt} ze zbioru *Kroki(plan)* zawierający
  warunek *c*, który nie został jeszcze osiągnięty
return S_{akt}, *c*

procedure WYBIERZ_OPERATOR(*plan,Operatory,*S_{akt}*,c*)

wybierz krok S_{nowy} ze zbioru *Operatory* lub *Kroki(plan)*,
  który ma jako efekt *c*.
if nie ma takiego kroku **than fail**
dodaj poł.przycz. $S_{nowy} \xrightarrow{c} S_{akt}$ do *Połączenia*(*plan*)
dodaj porządek $S_{nowy} \langle S_{akt}$ do *Porządki*(*plan*)
if S_{nowy} jest nowo dodanym krokiem ze zbioru *Operatory* **then**
  dodaj S_{nowy} do *Kroki*(*plan*)
  dodaj *Start* $\langle S_{nowy} \langle$ *Finisz* do *Porządki(plan)*

procedure ELIM_ZAGROŻENIA(*plan*)

for each S_{zagr}, który zagraża $S_i \xrightarrow{c} S_j$ w zbiorze *Połączenia(plan)* **do**
  wybierz jedną z:
    Promocja: dodaj $S_{zagr} \langle S_i$ do *Porządki(plan)*
    Degradacja: dodaj $S_j \langle S_{zagr}$ do *Porządki(plan)*
  if not KONSEKWENTNY(*plan*) **than fail**
end

Warto zauważyć, że POP jest algorytmem regresywnym, ponieważ zaczyna on działać mając cele, które należy osiągnąć, a następnie cofa się by znaleźć operatory, które te cele osiągną. Jeśli wszystkie warunki każdego kroku zostaną osiągnięte, wówczas algorytm znalazł rozwiązanie. POP jest poprawny i kompletny; każdy zwrócony przez niego plan jest poprawnym rozwiązaniem, jeśli istnieje rozwiązanie problemu, to algorytm go znajdzie.

6.7 Planowanie przy częściowo zainicjowanych operatorach

W poprzedniej sekcji przedstawiony został algorytm, który jednak posiadał kilka elementów, które zostały w nim pominięte. W szczególności, nie brał on w ogóle pod uwagę powiązań zmiennych. W większości przypadków musimy jednak śledzić listę powiązań zmiennych i unifikować odpowiednie wyrażenia we właściwym czasie.

Jest jedna istotna decyzja, którą należy podjąć: w procedurze ELIM_ZAGROŻENIA, czy operator, który posiada efekt $\neg W(x)$ powinien być traktowany jako zagrożenie dla warunku $W(dom)$? Aktualnie potrafimy już rozróżnić pomiędzy zagrożeniem i brakiem zagrożenia, lecz w tym przypadku powiemy o **możliwym zagrożeniu**. Są trzy sposoby rozwiązujące problem możliwych zagrożeń:

- **Rozwiąż teraz przy pomocy równości:** Sposób ten polega na zmodyfikowaniu procedury ELIM_ZAGROŻENIA tak, by zagrożenia były eliminowane kiedy tylko się pojawią. Dla przykładu: jeśli wybrany został operator, który jako efekt posiada $\neg W(x)$, by uchronić warunek $W(dom)$ zostanie przykładowo dodane przypisanie $x = SM$.
- **Rozwiąż teraz przy pomocy nierówności:** Sposób ten polega na rozszerzeniu języka stosowanego przy powiązaniach o element nierówności $\neq$. Dodając powiązanie $x \neq dom$ nie musieliśmy wybierać konkretnego miejsca dla x, zatem zastosowaliśmy mniejsze ograniczenie. Pojawia się jednak problem nieco bardziej skomplikowanej implementacji funkcji (np. Unifikacji), które dotychczas analizowały tylko równości.
- **Rozwiąż później:** W tym przypadku ignorujemy możliwe zagrożenia i zajmujemy się nimi dopiero wtedy, gdy staną się zagrożeniami pewnymi. Oznacza to, że ELIM_ZAGROŻENIA nie będzie traktował $\neg W(x)$ jako zagrożenia dla $W(dom)$. Dopiero w momencie, gdy zostanie dodane do planu powiązanie $x = dom$, wówczas pojawi się pewne zagrożenie, które algorytm będzie chciał wyeliminować przez promocję lub degradację. Sposób ten

ma przewagę, gdyż wprowadza małe ograniczenia dla planu, jednak trudniej jest stwierdzić, czy plan jest rozwiązaniem.

Poniżej przedstawiona jest modyfikacja standardowego algorytmu POP przez drobne zmiany w procedurze WYBIERZ_OPERATOR i ELIM_ZAGROŻENIA, która uwzględnia technikę „Rozwiąż Później".

procedure WYBIERZ_OPERATOR(*plan,Operatory,*S_{akt}*,c*)

wybierz krok S_{nowy} ze zbioru *Operatory* lub *Kroki(plan)*, posiadający efekt c_{nowy} taki, że u = UNIFY*(c,* c_{nowy}*, Powiązania(plan))*

if nie ma takiego kroku **than fail**
dodaj u do *Powiązania(plan)*
dodaj poł.przycz. $S_{nowy} \xrightarrow{c} S_{akt}$ do *Połączenia*(*plan*)
dodaj porządek $S_{nowy} \langle S_{akt}$ do *Porządki*(*plan*)
if S_{nowy} jest nowo dodanym krokiem ze zbioru *Operatory* **then**
dodaj S_{nowy} do *Kroki*(*plan*)
dodaj *Start* $\langle S_{nowy} \langle$ *Finisz* do *Porządki(plan)*

procedure ELIM_ZAGROŻENIA(*plan*)

for each $S_i \xrightarrow{c} S_j$ w *Połączenia(plan)* **do**
for each Szagr w *Kroki(Plan)* **do**
for each *c'* w *Efekty(Szagr)* **do**
if ZAM(*Powiązania(plan),c*) = ZAM(*Powiązania(plan*),¬c)
then wybierz jedną z:
Promocja: dodaj $S_{zagr} \langle S_i$ do *Porządki(plan)*
Degradacja: dodaj $S_j \langle S_{zagr}$ do *Porządki(plan)*
if not KONSEKWENTNY(*plan*) **than fail**
end
end
end

Jeśli w planie pojawiają się operatory częściowo zainicjowane, należy zmodyfikować kryterium akceptujące plan jako rozwiązanie. Dotychczas rozwiązaniem był taki plan częściowy, którego linearyzacje zawsze osiągały cel końcowy. W przypadku częściowo zainicjowanych operatorów musimy także zapewnić, że wszystkie możliwe podstawienia osiągną cel końcowy. Rozszerzymy więc definicję osiągnięcia warunku przez krok:

Krok S_i **osiąga** warunek c kroku S_j jeśli: (1) $S_i \langle S_j$ i S_i posiada efekt, który koniecznie unifikuje się z c; i (2) nie ma takiego kroku S_k, że $S_i \langle S_k \langle S_j$ w jakiejś linearyzacji planu i S_k posiada efekt, który mógłby się unifikować z $\neg c$.

Algorytm POP może być widziany jako konstrukcja dowodu, że każdy warunek kroku docelowego jest osiągalny. Procedura WYBIERZ_OPERATOR wybiera z puli operatorów te, które spełniają własność (1), natomiast ELIM_ZAGROŻENIA upewnia się, czy spełniona jest własność (2) dokonując promocji lub degradacji możliwych zagrożeń.

Stosując technikę „Rozwiąż Później" mamy jednak świadomość, że istnieją możliwe zagrożenia, które nie zostaną od razu wyeliminowane. Potrzebujemy więc jakiegoś mechanizmu, by upewnić się, czy zanim oddamy gotowy plan, te wszystkie możliwe zagrożenia zostaną wyeliminowane. Okazuje się, że jeśli stan początkowy nie zawiera zmiennych, a każdy operator posiada nadmienia o wszystkich swoich zmiennych w swoich warunkach, wtedy kompletny plan wygenerowany przez POP będzie zawsze w pełni zainicjowany. W przeciwnym wypadku musimy zmodyfikować funkcję ROZWIĄZANIE? by sprawdzała czy nie ma niezainicjowanych zmiennych i w razie czego znajdowała dla nich powiązania. Jeśli to jest spełnione, wtedy rozwiązanie dane przez POP jest poprawne.

Trudniej jest udowodnić, czy POP jest algorytmem kompletnym. Ogólnie rzecz biorąc możemy przyjąć, że generuje on wszystkie możliwe plany spełniające własność (1), a następnie filtruje je, odrzucając plany, które nie spełniają własności (2). Stąd, jeśli istnieje plan, spełnia on własności (1) i (2), a algorytm POP go znajdzie.

6.8 Inżynieria Wiedzy dla problemu planowania

Metodologia przy rozwiązywaniu problemów planowania jest następująca:

- Decyzja o czym będzie mowa
- Decyzja odnośnie języka wyrażeń, operatorów i obiektów
- Zakodowanie operatorów dla dziedziny
- Zakodowanie opisu problemu
- Przekazanie problemów jednostkom planującym i odebranie wyników.

Wszystkie z tych 5 części przedstawimy na przykładzie:
Świat klocków

O czym będzie mowa: Pokażemy teraz jak zdefiniować wiedzę dla klasycznego zagadnienia planowania – Świata Klocków. Dziedzina problemu składa się ze zbioru sześciennych bloków leżących na stole. Bloki mogą być dowolnie układane, lecz na dowolnym bloku może znajdować się bezpośrednio tylko jeden inny blok. Ramie robota może podnieść blok i przenieść go na inną

pozycję (albo na inny blok, albo na stół). Ramie może w danej chwili trzymać tylko jeden blok, zatem nie może przenosić bloku, który ma na sobie inny blok. Celem będzie za każdym razem zbudowanie jednego lub kilku stosów z dostępnych bloków, np. dla klocków A, B, C, D, E celem może być stos A/C/E oraz B/D.

Język: Obiektami w tej dziedzinie są bloki oraz stół. Są one reprezentowane przez stałe. Użyjemy predykatu *Na(b, x)* by zapisać, że blok *b* jest na obiekcie x (x jest albo innym blokiem, albo stołem). Operator odpowiedzialny za przemieszczenie bloku b z miejsca znajdującego się na x do miejsca znajdującego się na y będzie miał postać *Ruch(b, x, y)*. Pamiętamy, że jednym z warunków do przemieszczenia bloku *b* jest, by na b nie leżał żaden blok. W logice pierwszego stopnia wyrazilibyśmy to jako: $\neg\exists x\ Na(x, b)$ lub $\forall x\ \neg Na(x, b)$. Nasz język nie pozwala jednak na użycie takich form, dlatego należy wymyślić coś innego, np. predykat, który mówi, że na danym bloku nic się nie znajduje. Będzie nim *Czysty(x)*, co oznacza, że nie ma niczego na x.

Operatory: Operator Ruch przemieszcza blok *b* z *x* na *y* jeśli zarówno *b* jak i *y* są czyste. W efekcie tej operacji *x* staje się czyste, natomiast *y* przestaje być czyste. Formalny opis operacji *Ruch* jest następujący:

Op(AKCJA: *Ruch(b,x,y)*,
WARUNEK: $Na(b,x) \wedge Czysty(b) \wedge Czysty(y)$,
EFEKT: $Na(b,y) \wedge Czysty(x) \wedge \neg Na(b,x) \wedge \neg Czysty(y)$)

Niestety, ten operator nie zachowuje poprawnej własności *Czysty*, gdy *x* lub *y* jest stołem. Gdy *x= stół*, operator ma efekt *Czysty(stół),* podczas gdy *stół* nie powinien stać się czysty. Podobnie gdy *y = stół,* operator posiada warunek *Czysty(stół)*, podczas gdy *stół* nie musi być czysty by móc na niego położyć blok. By rozwiązać ten problem zrobimy dwie rzeczy. Na początku wprowadzimy kolejny operator by przemieścić blok *b* z *x* na *stół*.

Op(AKCJA: *RuchNaStół(b,x)*,
WARUNEK: $Na(b,x) \wedge Czysty(b)$,
EFEKT: $Na(b, stół) \wedge Czysty(x) \wedge \neg Na(b,x)$)

Ponadto wprowadzimy następującą interpretację *Czysty(x)*: „nad *x* jest czysto, można tam położyć blok”. Wedle tej interpretacji *Czysty(stół)* będzie zawsze częścią sytuacji początkowej, poza tym naturalnym jest, że *Ruch(b,stół,y)* posiada jako efekt *Czysty(stół)*. Jedynym problemem jest to, że nic nie powstrzymuje jednostki planującej do użycia operacji *Ruch(b,x,stół)* zamiast operacji *RuchNaStół(b,x)*. Możemy albo zaniechać ten problem, co doprowadzi nas jednak do niepoprawnych odpowiedzi, albo wprowadzić kolejny predykat *Blok* i dodać $Blok(x) \wedge Blok(y)$ do warunków operacji *Ruch.*

Ostatecznie powinniśmy jeszcze zająć się nonsensownymi operacjami takimi jak Ruch(b,x,x), w wyniku których nic nie powinno się dziać. Często zajmując się planowaniem pomijamy takie sytuacje, co może doprowadzić do poważnych sprzeczności. By rozwiązać ten problem musielibyśmy tak naprawdę wprowadzić nierówności dla warunków operacji: b≠x≠y.

6.9 Dekompozycja Hierarchiczna

Omawiany w tym rozdziale przykład z „zakupami" dał rozwiązanie na dosyć dużym poziomie abstrakcji. Plan taki jak:

[*Idź(Supermarket), Kup(miód), Kup(ziemniaki), Idź(dom)*]

jest dobrym opisem wysokiego poziomu co należy zrobić, ale daleko mu do instrukcji, które byłyby przetwarzane bezpośrednio przez fizyczne mechanizmy agenta wybierającego się na zakupy. Stąd powyższy plan jest dla agenta niewystarczający do wykonania jakiejkolwiek czynności. Z drugiej strony poniższy plan niskiego poziomu:

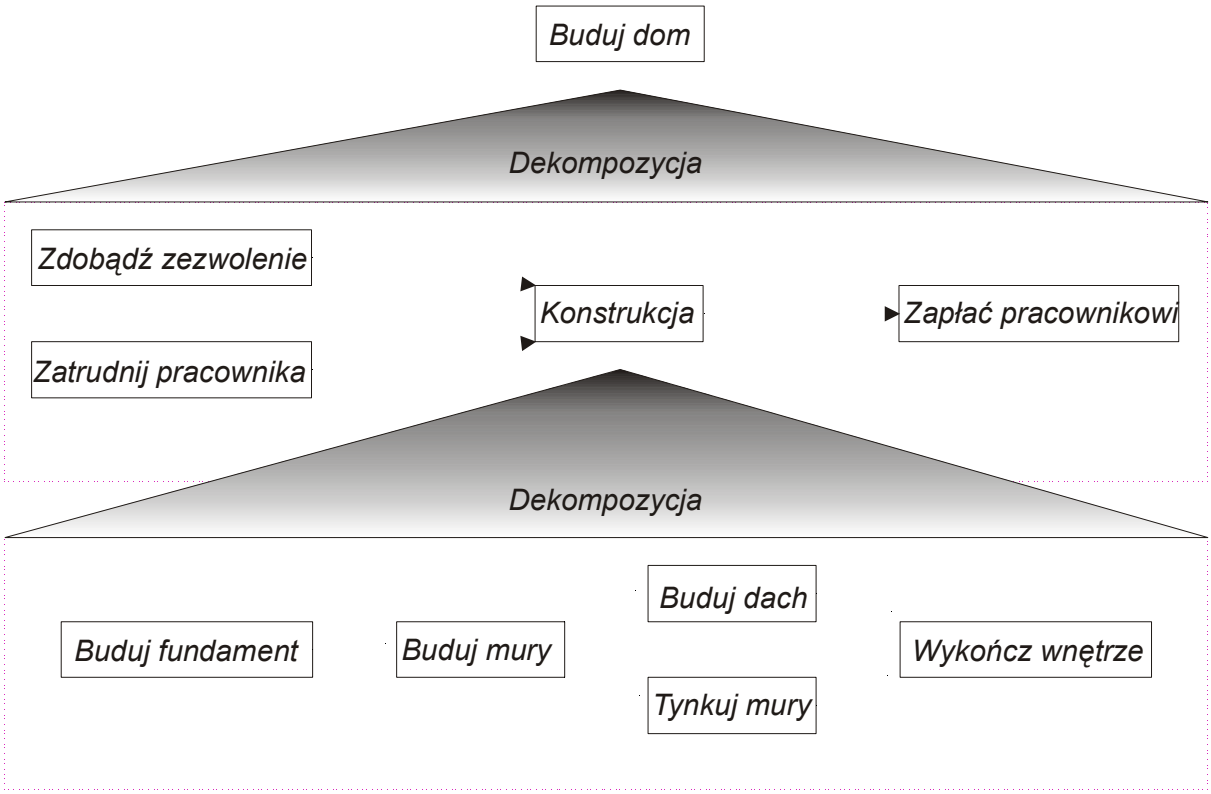
[*Wprzód(1metr),ObrótWPrawo(40°), WTył(2metry) ...*]

musiałby zawierać tysiące kroków, by rozwiązać problem z „zakupami". Przestrzeń planów o takiej ilości kroków jest tak olbrzymia, że przedstawiona metoda poszukiwania rozwiązania nie znalazłaby go w przyzwoitym czasie.

By rozwiązać ten problem stosuje się **dekompozycję hierarchiczną**, w której **abstrakcyjny operator** rozkładany jest na grupę kroków, które tworzą plan implementujący ten operator. Takie dekompozycje mogą być zapamiętywane w bibliotece planów i zastosowane w miarę potrzeb.

Rozpatrzmy problem budowy domu. Abstrakcyjny operator Buduj(dom) może zostać rozłożony na plan składający się z 4 kroków: Zdobądź pozwolenie, Zatrudnij pracownika, Konstrukcja, Zapłać pracownikowi, tak jak jest to pokazane na rysunku nr 20. Kroki tego planu mogą podlegać dalszej dekompozycji i tak: Konstrukcja rozkłada się na plan złożony z kroków: Buduj fundament, Buduj mury, Buduj dach, Tynkuj mury, Wykończ wnętrze. Takie rozkłady moglibyśmy kontynuować aż każdy abstrakcyjny operator rozłożony zostanie do **operatorów prymitywnych** bezpośrednio wykonywanych przez agenta (np. wbij gwóźdź do ściany). Dopiero gdy plan składa się z samych operatorów prymitywnych mówimy, że jest on kompletny.

By dekompozycja hierarchiczna zaczęła działać pozostaje nam jeszcze: (1) Rozszerzyć język STRIPS o operatory nie-prymitywne, oraz (2) zmodyfikować algorytm planujący, by pozwalał na zamianę operatorów abstrakcyjnych na ich dekompozycje.



Rysunek 20 Hierarchiczna dekompozycja przy budowie domu

Rozszerzanie języka:

Na rozszerzanie języka składają się dwie czynności.

Na początku musimy określić jakie operatory są dla agenta prymitywne, a jakie nie-prymitywne. Podział ten jest względny i zależy od rodzaju agenta i wykonywanego przez niego planu. Przykładowo dla kierownika budowy, operacją prymitywną będzie wstawienie okna. Czynność ta będzie dla zwykłego robotnika budowlanego operacją abstrakcyjną, którą rozłoży na plan składający się odpowiednich dla niego operacji prymitywnych np. umocowanie szyby w ramie. Drugą czynnością jest dodanie zbioru metod dekompozycji. Każda metoda jest postaci DEKOMPOZYCJA(*o,p*), co oznacza, że nie-prymitywny operator unifikowany z *o* może zostać rozłożony do planu *p*. Poniżej przedstawiona jest dekompozycja operatora Konstrukcja:

DEKOMPOZYCJA(Konstrukcja,
Plan(KROKI:{S_1: Buduj(*Fundament*), S2: Buduj(*Mury*),
S_3: Buduj(*Dach*), S4: Tynkuj(*Mury*)
S_5: Wykończ(*Wnętrze*)}
PORZĄDKI: {$S_1 \langle S_2 \langle S_3 \langle S_5$, $S_2 \langle S_4 \langle S_5$},
POWIĄZANIA: {},
POŁĄCZENIA: { $S_1 \xrightarrow{Fundament} S_2$, $S_2 \xrightarrow{Mury} S_3$, $S_2 \xrightarrow{Mury} S_4$,
$S_3 \xrightarrow{Dach} S_5$, $S_4 \xrightarrow{Otynkowany} S_5$ }))

Pozostaje nam upewnić się, czy dekompozycja jest poprawną implementacją operatora. Powiemy, że plan *p* poprawnie implementuje operator *o,* jeśli jest to poprawny i konsekwentny plan, który pozwala na osiągnięcie efektów operatora *o*, przy warunkach początkowych operatora *o*.

- *p* musi być konsekwentny (brak sprzeczności w PORZĄDKACH i POWIĄZANIACH występujących w planie *p*),
- każdy efekt operatora *o* musi być wyrażony przynajmniej przez jeden krok planu *p* (i nie usunięty przez jakiś następny krok tego planu),
- Każdy warunek każdego z kroków planu *p* musi być osiągnięty przez krok planu *p* lub przez warunek operatora *o*.

To gwarantuje nam możliwość zastąpienia operatorów abstrakcyjnych przez odpowiednie plany powstałe w wyniku dekompozycji.

Modyfikacja algorytmu

W celu napisania algorytmu wykorzystującego dekompozycję hierarchiczną, którego nazwiemy HD_POP, wprowadzimy kilka modyfikacji do istniejącego już algorytmu POP. Są dwie podstawowe różnice.

Po pierwsze, algorytm powinien nie tylko starać się znaleźć drogę do osiągnięcia nieosiągniętych dotychczas warunków planu, ale również potrafić rozkładać operatory nie-prymitywne. Oba te warunki muszą zostać spełnione w celu uzyskania kompletnego, prymitywnego planu, przy czym ich kolejność nie gra roli przy back-trackingu algorytmu. HD_POP wykonuje każdą z tych dwóch czynności oddzielnie.

Po drugie, algorytm na wejściu otrzymuje *plan*, a nie sam *cel*. Nie jest to dużą przeszkodą, ponieważ cel może być reprezentowany jako plan *Start-Finisz*. Otrzymujemy przez to kompatybilność ze standardowym algorytmem POP. Algorytm HD_POP wygląda następująco:

function HD_POP(*plan, Operatory, Metody*) **returns** *plan*
inputs: abstrakcyjny *plan* zawierający kroki *Start*, *Finisz*, oraz inne.

loop do
  if ROZWIĄZANIE?(*plan*) **then return** *plan*
  Sakt,c←WYBIERZ_PODCEL(*plan*)
  WYBIERZ_OPERATOR(*plan,Operatory,*S_{akt}*,c*)
  $S_{nie\text{-}prym}$ ← WYBIERZ_NIEPRYMITYWNY(*plan*)
  WYBIERZ_DEKOMPOZYCJE(*plan*, *Metody*, $S_{nie\text{-}prym}$)
  ELIM_ZAGROŻENIA(*plan*)
end

Widać, że algorytm najpierw stara się wybrać operator, który osiąga dotychczas nieosiągnięty cel, a w dalszej części znajduje nie-prymitywny operator i szuka dla niego dekompozycji.

By HD_POP działał, musimy jeszcze zmodyfikować ROZWIĄZANIE? w taki sposób, by sprawdzać czy każdy krok planu jest prymitywny. Są dwie nowe procedury: WYBIERZ_NIEPRYMITYWNY arbitralnie wybiera nieprymitywny krok z planu. Funkcja WYBIERZ_DEKOMPOZYCJE wybiera metodę dekompozycji i stosuje ją. Jeśli do dekompozycji kroku $S_{nie\text{-}prym}$ użyta jest metoda *„md”*, wówczas własności planu są zmienione w następujący sposób:

- KROKI: Dodaj do *planu* wszystkie kroki *md*, usuń krok $S_{nie\text{-}prym}$.
- POWIĄZANIA: Dodaj do *planu* wszystkie powiązania metody *md*. Jeśli to prowadzi do sprzeczności, sygnalizuj „**fail**”
- PORZĄDKI: Zamień każde wyrażenie porządkowe postaci $S_a \langle S_{nie\text{-}prym}$ na wyrażenia porządkowe ustawiające S_a przed ostatnimi krokami *md*. Innymi słowy, jeśli S_m jest krokiem *md*, i nie ma żadnego S_j w *md* takiego, że $S_m \langle S_j$, wtedy dodaj wyrażenie $S_a \langle S_m$. Podobnie, zamień każde wyrażenie postaci $S_{nie\text{-}prym} \langle S_z$ na wyrażenia, które ustawiają S_z po najwcześniejszych krokach *md*. Następnie wywołujemy ELIM_ZAGROŻENIA by dodać dodatkowe porządki, które mogą być konieczne.
- POŁĄCZENIA: Jeśli w *planie* było połączenie przyczynowe $S_i \xrightarrow{C} S_{nie\text{-}prym}$, zamień je z zestawem połączeń postaci $S_i \xrightarrow{C} S_m$, gdzie S_m jest krokiem *md*, posiadającym jako warunek *c* i nie ma żadnego wcześniejszego kroku w *md*, który posiada jako warunek *c*. (Jeśli jest wiele takich kroków, które posiadają warunek *c*, dodaj połączenie przyczynowe dla każdego. Jeśli nie ma takich kroków, wówczas połączenie wychodzące od S_i może zostać usunięte, ponieważ *c* było niekoniecznym warunkiem dla $S_{nie\text{-}prym}$). Podobnie, dla każdego połączenia $S_{nie\text{-}prym} \xrightarrow{C} S_j$ na *planie,* zamień je z zestawem połączeń $S_m \xrightarrow{C} S_j$, gdzie S_m jest krokiem *md*, który posiada jako efekt *c* i nie ma w *md* wcześniejszego kroku posiadającego jako efekt *c*.

6.10 Analiza Dekompozycji Hierarchicznej

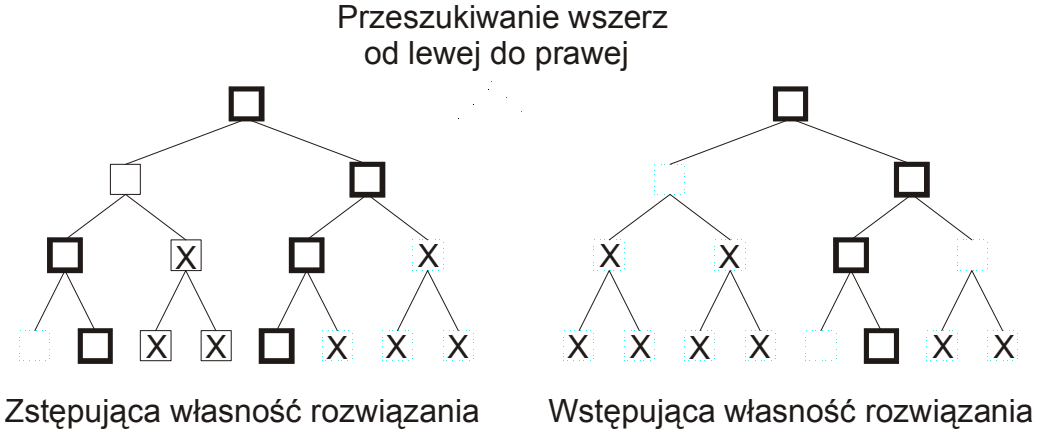
Dekompozycja wydaje się być bardzo dobrym rozwiązaniem, analogicznym do znanych wszystkim makr występujących w programowaniu. Pozwala ona na tworzenie planu z klocków, którymi są operatory abstrakcyjne, które s kolei mogą się rozkładać aż do operatorów prymitywnych. W tej sekcji przyjrzymy się dokładniej tej intuicyjnej idei i przeanalizujemy efektywność i skuteczność jej działania.

Załóżmy, że znaleźliśmy rozwiązanie, które jest planem składającym się z operatorów abstrakcyjnych. Będziemy je odtąd nazywali **rozwiązaniem abstrakcyjnym**. Znajdowanie pojedynczego rozwiązania abstrakcyjnego nie jest trudne (niewielka ilość kroków), a kontynuując ten proces otrzymamy w końcu plan prymitywny bez znacznego używania back-trackingu.

Oznacza to, że poszukując planów abstrakcyjnych, możemy znacznie zredukować drzewo poszukiwań, jeśli uznamy za prawdziwe poniższe własności:

- jeśli *p* jest rozwiązaniem abstrakcyjnym, wówczas istnieje prymitywne rozwiązanie, którego abstrakcją jest *p*. Jeśli ta własność jest prawdziwa, wtedy wraz ze znalezieniem rozwiązania abstrakcyjnego, możemy odrzucić z drzewa poszukiwań wszystkie inne abstrakcyjne plany. Jest to **zstępująca własność rozwiązania**.
- Jeśli abstrakcyjny plan jest niekonsekwentny, wówczas nie ma prymitywnego rozwiązania, którego abstrakcją jest ten plan. Jeśli ta własność jest prawdziwa, możemy odrzucić wszystkie plany będące dekompozycjami niekonsekwentnego planu abstrakcyjnego. Nazwiemy to **wstępująca własność rozwiązania**, gdyż oznacza ona też, że wszystkie abstrakcje rozwiązania prymitywnego są rozwiązaniami abstrakcyjnymi.

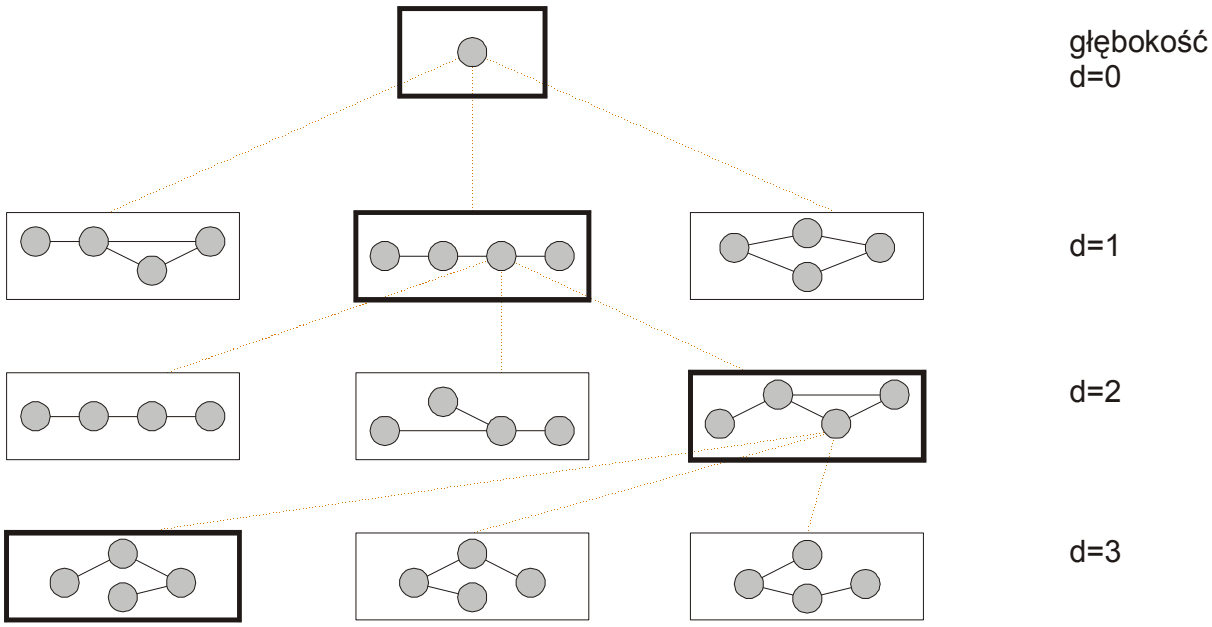
Rysunek nr 21 przedstawia oba te pojęcia graficznie. Każdy kwadrat reprezentuje cały plan (nie pojedynczy krok), a każdy łuk przedstawia dekompozycję pewnego kroku. Na samej górze jest plan abstrakcyjny, a na dole są plany z operatorami prymitywnymi. Pogrubione kwadraty oznaczają rozwiązanie (może być ono abstrakcyjne), kwadraty narysowane linią przerywaną oznaczają plany niekonsekwentne, a kwadraty z „X" w środku oznaczają plany, które według jednej z 2 własności nie muszą być analizowane przez algorytm.



Rysunek 21 Zstępująca i wstępująca własność rozwiązania

Obliczymy teraz jaka jest przewaga stosowania dekompozycji jeśli chodzi o czas znajdowania możliwego rozwiązania. Załóżmy, że jest co najmniej jedno rozwiązanie zawierające „n" prymitywnych kroków, a czas eliminowania zagrożeń oraz zarządzania powiązaniami i połączeniami zaniedbujemy: będzie nas interesować tylko czas wyboru odpowiednich kroków. Na rysunku nr 22 pokazana jest przestrzeń poszukiwań i dekompozycja hierarchiczna:

b=3 Ilość metod dekompozycji na krok
s=4 Ilość kroków w metodzie dekompozycji
d=3 Głębokość planu hierarchicznego



Rysunek 22 Wycinek przestrzeni poszukiwań przy dekompozycji hierarchicznej

Algorytm planujący nie korzystający z dekompozycji musiałby wygenerować n – krokowy plan, wybierając dla każdego kroku jedną z „b" możliwości (Zakładamy że ilość dekompozycji „b" nie-prymitywnego kroku jest taka sama, jak ilość nowych operatorów, które mogą osiągnąć jakiś warunek prymitywnego kroku). Wobec tego dostajemy w najgorszym przypadku złożoność obliczeniową $O(b^n)$.

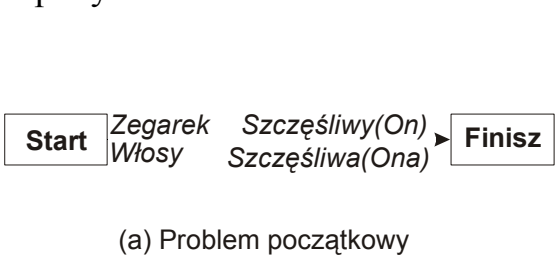
Przy planowaniu hierarchicznym możemy stosować strategię, która szuka tylko wśród tych dekompozycji, które dadzą abstrakcyjne rozwiązanie. W naszym powyższym modelu tylko 1 spośród b dekompozycji daje rozwiązanie. Algorytm planujący musi patrzyć na s·b kroków na głębokości d=1. Na głębokości d=2 patrzy na następne $s \cdot b$ kroków dla każdego kroku, który rozkłada, ale musi rozłożyć tylko 1/b z nich, stąd $b \cdot s^2$ kroków. Wobec tego całkowita ilość rozpatrywanych planów, to:

$$\sum_{i=1}^{d} bs^i = O(bs^d)$$

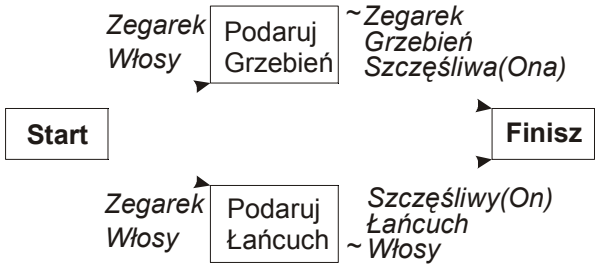
Dla przykładu z rysunku nr 22, algorytm wykorzystujący hierarchię musi rozpatrzyć 3×10^{30} planów, a hierarchiczny zaledwie 576.

Zstępująca i wstępująca własność rozwiązania wydaje się być czymś bardzo sinym i użytecznym. Na pierwszy rzut oka własności te mogą się wydawać naturalnymi elementami poprawnej dekompozycji. W rzeczywistości żadna własność nie ma gwarancji poprawności. Bez tych własności algorytm hierarchiczny może działać w najgorszym przypadku nawet tak długo jak algorytm nie korzystający z hierarchii.

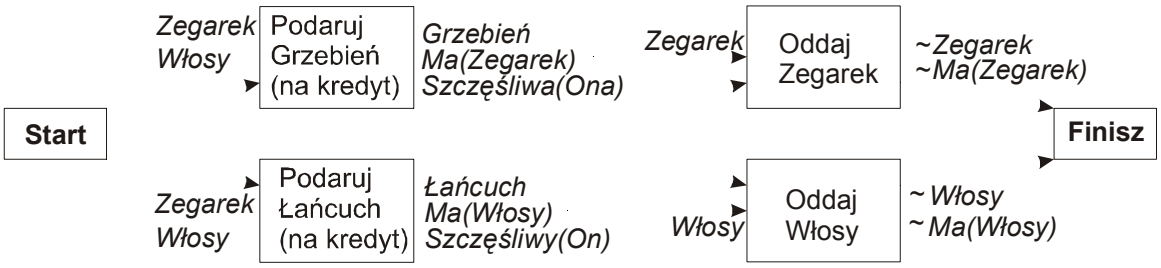
Na rysunku nr 23 pokazany jest przykład, gdy wstępująca własność rozwiązania okazuje się błędna. Oznacza to, że rozwiązanie abstrakcyjne jest niekonsekwentne, ale istnieje dekompozycja, która eliminuje ten problem. Załóżmy, że biedna para posiada tylko dwa wartościowe przedmioty: Ona posiada piękne długie włosy, a on złoty zegarek. Oboje planują kupić prezent, by uczynić drugą osobę szczęśliwą. On decyduje się sprzedać swój złoty zegarek i kupić dla niej zmysłowy grzebień, podczas gdy ona postanawia sprzedać swoje włosy, by kupić złoty łańcuch dla jego zegarka. Jak widać na rysunku 23(b) abstrakcyjny plan nie jest konsekwentny, jakkolwiek istnieje pewna metoda dekompozycji (Rysunek 23(c)), dzięki której plan staje się konsekwentny. Metoda ta polega na dekompozycji kroku Daj Grzebień na dwa etapy. Początkowo On decyduje się kupić grzebień „Na kredyt" obiecując, że dostarczy swój zegarek później. Drugim krokiem jest późniejsze oddanie przez niego zegarka. Podobnie postępuje Ona kupując dla niego złoty łańcuch i obiecując za niego zapłacić w późniejszym terminie. Przez zapłatą zarówno On jak i Ona otrzymują prezenty i są szczęśliwi, czego nie zakłócają późniejsze zapłaty.



(a) Problem początkowy



(b) Abstrakcyjny, niekonsekwentny plan



(c) Dekompozycja(b) dająca konsekwentne rozwiązanie

Rysunek 23 Problem obdarowywania prezentami

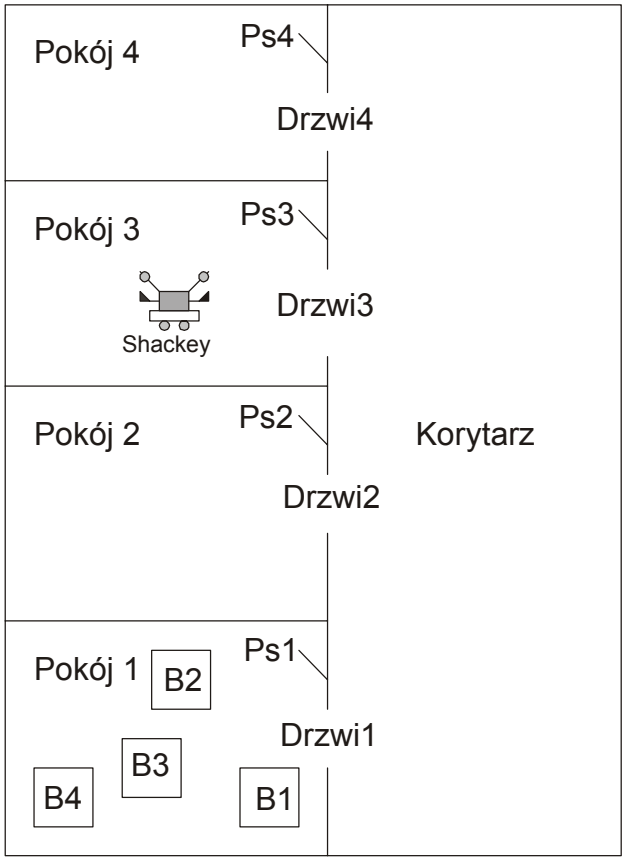
Jednym sposobem, by wstępująca własność rozwiązania była spełniona jest upewnienie się, czy każda metoda dekompozycji spełnia warunek **unikalnej podakcji**, który mówi, że istnieje krok w rozłożonym planie do którego dołączone są wszystkie warunki i efekty operatora abstrakcyjnego.

Czasem warto przeanalizować metody dekompozycji przed ich użyciem i sprawdzić, czy wszystkie spełniają warunek unikalnej podakcji. Dopiero to pozwala nam na swobodne obcinanie gałęzi poszukiwań i co za tym idzie, na efektywne wykorzystanie dekompozycji hierarchicznej.

6.11 Zadania

1. Załóżmy, że dysponujemy robotem o nazwie Shakey, który jest w świecie złożonym z 4 pokoi połączonych korytarzem; do każdego pokoju prowadzą drzwi; w każdym pokoju jest przełącznik światła (Ps) , tak jak na Rysunku nr 24 . Shakey może się poruszać z jednego miejsca do innego, popychać skrzynie (B1,B2,B3,B4), wchodzić na skrzynie, włączać i wyłączać światło. Jego akcjami są:

a) *Idź(x,y)* – przejście z jednej lokalizacji do drugiej. Jednym z warunków przejścia jest by zachodziło *W(Shakey,x)*.
b) *Pchaj(o,x,y)*. – Pchanie obiektu b z miejsca x do y. Musi zachodzić *Ruchomy(o)*.
c) *Wejdź(b)* – Powoduje wejście na skrzynię b (Shakey musi być na ziemi koło skrzyni; Na skrzynię b musi dać się wejść – Parametr *Niska(b)*.
d) *Zejdź(b)* – Przeciwieństwo akcji *Wejdź(b)*.
e) *Zapal(Ps)* – Zapala światło w pomieszczeniu. Warunek: przełącznik jest wysoko, więc Shakey może go dosięgnąć tylko, gdy jest na skrzyni znajdującej się koło przłącznika.
f) *Zgaś(Ps)* – Przeciwieństwo akcji *Zapal(Ps)*.



Rysunek 24 Świat Shakey'a

Zadaniem jest:

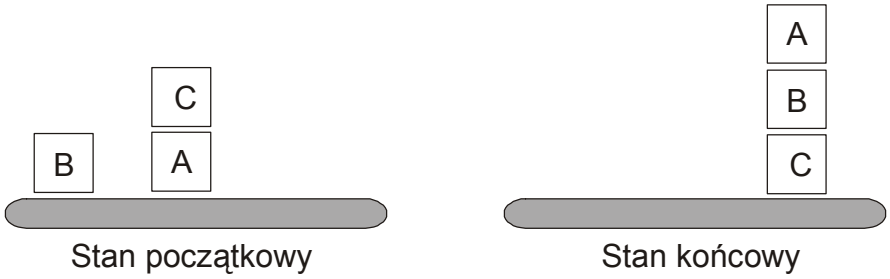
(i) Dla wszystkich 6 akcji dodaj brakujące predykaty (np: mówiące, że światło 3 jest zgaszone), a następnie opisać świat, w którym jest Shakey na Rysunku nr 24.

(ii) Opisz akcje Shakey'a w języku STRIPS

(iii) Napisz plan dla Shakeya, by przeniósł on Skrzynię B2 do Pokoju 4 (Start: Sytuacja z Rysunku nr 24).

(iv) Niech na początku wszystkie światła są zgaszone. Napisz plan, by Shakey je wszystkie pozapalał (Zaczyna w sytuacji z Rysunku nr 24).

2. Na rysunku nr 25 pokazany jest problem świata bloków nazywany Anomalią Sussman'a. Zapisz go przy użyciu języka STRIPS, a następnie użyj algorytmu POP, by osiągnąć stan końcowy.



Rysunek 25 Problem Blokowy

3. Analiza problemu małpy i bananów. Małpa znajduje się w pokoju w miejscu A, skrzynia w miejscu B, a banany wiszą w miejscu C. Małpa by dosięgnąć bananów musi wejść na skrzynię, która znajduje się w miejscu, gdzie są banany. Małpa może iść z jednego miejsca do drugiego, pchać skrzynię z jednego miejsca do drugiego, wejść (ew. zejść) na skrzynię i sięgać po banany (jeśli są w zasięgu), w wyniku czego zaczyna je posiadać.

(i) Opisz sytuację początkową używając odpowiednich predykatów.
(ii) Opisz w języku STRIPS akcje, które małpa może wykonać.
(iii) Napisz plan, pozwalający małpie na zdobycie bananów, a następnie ustawienie skrzyni tam, gdzie była na początku.

7 Nauczanie

We wszystkich poprzednich rozdziałach za „inteligencję” uważaliśmy wiedzę, która została agentowi przekazana przez projektanta systemu. Nietrudno wywnioskować, że jeśli taki agent znajdzie się w nieznanym mu środowisku, będzie skazany na niepowodzenie. Za każdym razem, gdy projektant nie posiada kompletnej wiedzy o otoczeniu, agent będzie zdany sam na siebie, toteż będzie musiał się uczyć. To pozwoli mu na osiągnięcie pewnej autonomii, uniezależnienia się od ścisłej kontroli projektanta.

Nauczanie na podstawie obserwacji

W tej sekcji skupimy się na tym jak agent może się uczyć na podstawie obserwacji otoczenia, czyli informacji, które do niego napływają z zewnątrz. Dotychczas każdy agent, którym się zajmowaliśmy posiadał **element decyzyjny**, który był odpowiedzialny za każde zachowanie agenta. Ponadto, wszystkie bodźce z zewnątrz dostarczały informacje, które były wykorzystywane tylko przez omawiany element decyzyjny. Działał on na zasadzie algorytmu komputerowego, którego dane pochodziły z otoczenia.

Wyposażymy teraz naszego agenta w **element nauczania**, którego głównym zadaniem będzie udoskonalanie działania elementu decyzyjnego. Nauczanie będzie realizowane w głównej mierze dzięki wzorcom pochodzącym z zewnątrz. Im więcej wzorców dotrze do elementu nauczania, tym ważniejsze i dokładniejsze hipotezy będzie on podejmował.

Każdy algorytm nauczania będzie na wejściu otrzymywał właśnie wzorce, a hipotezy będące rezultatem jego działania będą uogólnionymi prawami dotyczącymi otoczenia. Każda poprawna hipoteza będzie musiała zgadzać się gdy przetestujemy ją na wzorcach, które ją zbudowały.

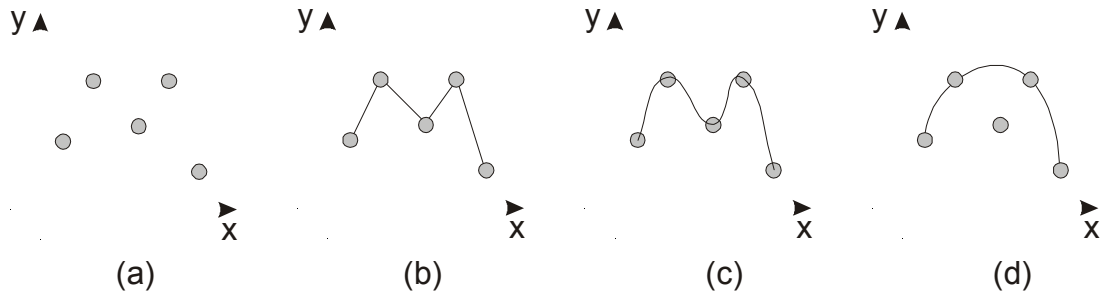
Ważnym zagadnieniem w nauczaniu jest wiedza początkowa, którą agent posiada jeszcze przez rozpoczęciem procesu nauczania. Człowiek w dużej mierze zawdzięcza swoją zdobytą wiedzę otoczeniu, w którym się znajduje i uczy. Niektórzy psychologowie twierdzą, że nawet noworodki dziedziczą część wiedzy, która zdecydowanie przyspiesza późniejsze nauczanie. Jest bez wątpienia, iż potrafią one widzieć, oddychać, chwytać przedmioty i rozpoznawać tony różnej wysokości, i że nikt ich tego przecież nie nauczył. Jeden fakt możemy jednak stwierdzić bez obaw. Wcześniejsza wiedza zdecydowanie przyspiesza proces nauczania.

7.1 Nauczanie indukcyjne

Nim zaczniemy zajmować się tym typem nauczania warto napomnieć, że w każdym nauczaniu chodzi o to, by agent zdobył umiejętność określania wartości różnych funkcji matematycznych. Każdy problem może zostać przedstawiony jako pewna funkcja. Przykładowo jeśli agent będzie miał samodzielnie odpowiedzieć na pytanie, czy pod wpływem określonych *warunków* ma wykonać jakąś czynność, możemy ten problem sprowadzić do funkcji binarnej (1 – wykonaj czynność, 0 – nie wykonuj czynności), która jako parametry przyjmuje wyżej wymienione *warunki*.

Omówimy na początku taki typ uczenia agenta, w którym element nauczania otrzymuje zestaw wartości funkcji dla różnych (dowolnych) argumentów, a następnie próbuje znaleźć funkcję, która aproksymuje punkty o zadanych wartościach. Innymi słowy, element nauczania otrzymuje zestaw par (wzorców) (x,f(x)), gdzie x jest argumentem, a f(x) wartością funkcji. Zadaniem algorytmu jest znalezienie funkcji h, która jak najlepiej aproksymuje funkcję f. Funkcja h nazywana jest **hipotezą**.

Na rysunku nr 26 pokazany jest geometryczny przykład omawianego typu nauczania. Wzorcami są punkty (x,y) ; y=f(x) znajdujące się na rysunku nr 26(a). Celem algorytmu jest znalezienie funkcji h, która najlepiej dopasowuje te punkty. Pierwszą hipotezą (26(b)) jest funkcja przedziałami liniowa, która w pełni pokrywa się z pięcioma wzorcami. Drugą hipotezą jest bardziej skomplikowana „bardziej różniczkowalna" funkcja, która także uwzględnia wszystkie wzorce. W ostatniej hipotezie mamy funkcję, która ma mniejsze wahania kosztem pominięcia jednego, najbardziej odchylonego wzorca. Na tym etapie szukania hipotezy nie jesteśmy w stanie ocenić, która z (b), (c), (d), jest lepsza.



Rysunek 26 Nauczanie indukcyjne - przykład geometryczny

Wróćmy jednak do agentów. Załóżmy, że mamy do czynienia z agentem „krótkowzrocznym" (który podejmuje swoje decyzje bezpośrednio pod wpływem spostrzeżeń) uczonym przez nauczyciela.

W przedstawionym poniżej szkielecie algorytmu opisującego jego działanie występuje procedura $A_{krotkowzroczny}$EL_NAUCZANIA, która modyfikuje globalną zmienną *wzorce*, przechowującą pary (spostrzeżenie, akcja). Spostrzeżeniem może być np. układ figur na planszy do gry w Reversi, a akcją pole, na które postawimy naszą figurę. Gdy procedura $A_{krotkowzroczny}$EL_DECYZYJNY spotka się ze spostrzeżeniem (układem planszy), które już zna, wtedy wybiera odpowiadającą mu akcję. W przeciwnym wypadku wywołuje algorytm nauczania INDUKUJ przekazując jako parametry wzorce, które są już znane. INDUKUJ zwraca hipotezę h, którą agent używa do wyboru akcji.

```
global wzorce ← {}

function A_krotkowzroczny EL_DEZYCYJNY(spostrzeżenie) returns akcja
    if (spostrzeżenie,a) jest w zbierze wzorce then return a
    else
        h←INDUKUJ(wzorce)
        return h(spostrzeżenie)

procedure A_krotkowzroczny EL_NAUCZANIA(spostrzeżenie, akcja)
    inputs: spostrzeżenie, akcja – odebrane z elementu decyzyjnego

    wzorce ← wzorce ∪ {(spostrzeżenie, akcja)}
```

Jest wiele wariantów tego prostego schematu. Przykładowo agent mógłby realizować **nauczanie progresywne**: zamiast za każdym razem, gdy potrzebna jest nowa nieznana akcja stosować algorytm nauczania dla całego zestawu wzorców, agent może po prostu próbować modyfikować swoje stare hipotezy pod wpływem nowego wzorca. Ponadto, agent może otrzymać raport na temat jakości akcji, które wybiera.

Funkcja $A_{krotkowzroczny}$EL_DEZYCYJNY nie mówi nic o sposobie prezentacji hipotez. Z uwagi na swoją zrozumiałość i klarowność do algorytmów nauczania stosuje się często język logiki. W dalszej części rozdziału przedstawimy dwa sposoby nauczania dla zdań logicznych: oparte na **Drzewie Decyzyjnym** oraz na **Aktualnie Najlepszej Hipotezie.**

Wybór reprezentacji dla żądanej funkcji jest prawdopodobnie najważniejszym zagadnieniem, na które musi zwrócić uwagę projektant uczącego się agenta. Wybór ten nie tylko wpływa na naturę algorytmu nauczania, ale i na wykonywalność problemu.

7.2 Nauczanie przy pomocy drzew decyzyjnych

Drzewa decyzyjne są najprostszym i zarazem bardzo efektywnym sposobem nauczania agenta. Są one proste w implementacji i pozwalają na łatwe zrozumienie natury nauczania indukcyjnego. W tej sekcji skupimy się najpierw na elemencie decyzyjnym, wyjaśnimy jego budowę i działanie, a następnie przejdziemy do elementu nauczania.

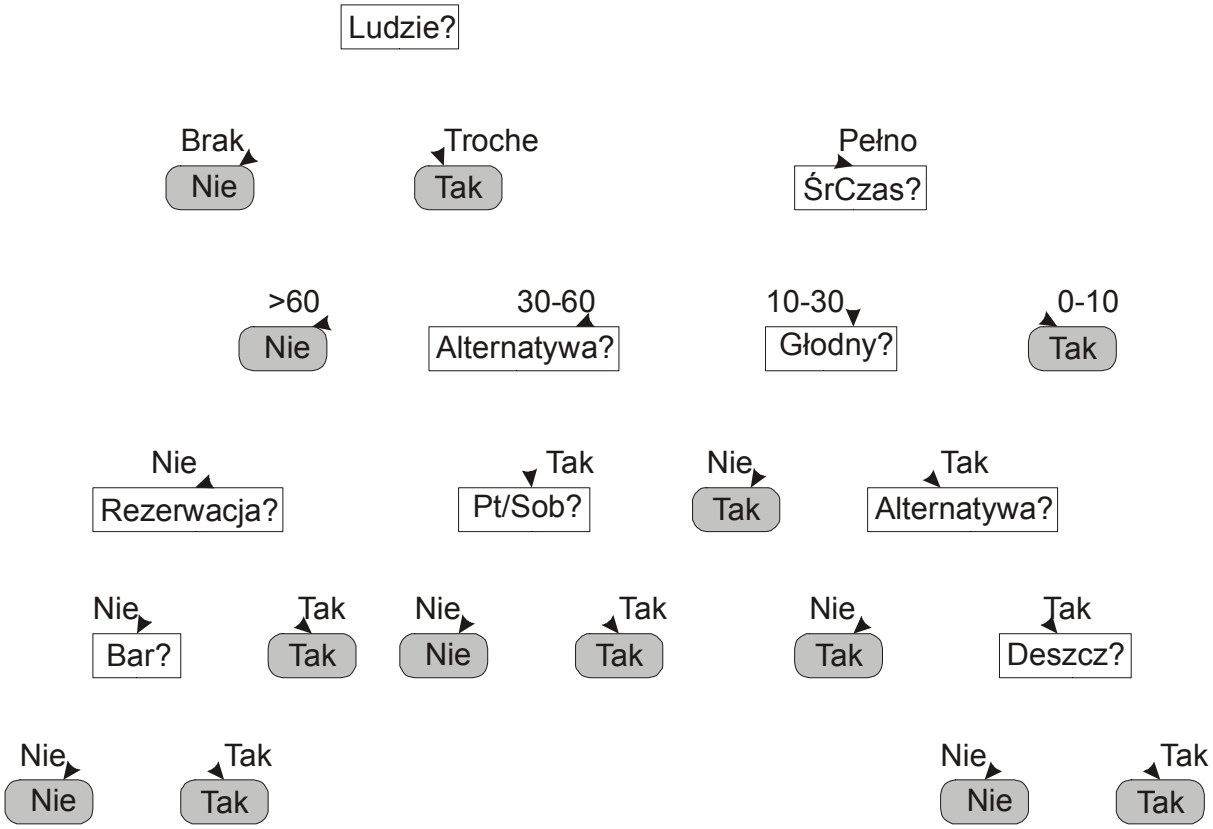
Element decyzyjny

Każde drzewo decyzyjne reprezentuje pewną funkcję. Mimo iż zbiór wartości może być duży, my ograniczymy się do funkcji binarnych, które w zupełności pozwolą nam rozwiązywać problemy, z którymi będzie się stykał nasz agent. Drzewo decyzyjne otrzymuje więc na wejściu obiekt składający się z zestawu parametrów, następnie analizuje go i zwraca rezultat, w tym przypadku decyzję tak / nie. Każdy wewnętrzny węzeł w drzewie decyzyjnym odpowiada za jeden z parametrów obiektu wejściowego. Gałęzie wychodzące z tego parametru są jego możliwymi wartościami. Każdy liść drzewa decyzyjnego przechowuje wartość logiczną, która zostanie zwrócona, jeśli liść zostanie osiągnięty.

W celu wyjaśnienia zasady funkcjonowania drzewa decyzyjnego przedstawimy problem oczekiwania na wolny stolik w restauracji. Celem będzie tu nauczenie agenta w jakich sytuacjach powinien czekać, a w jakich nie, co sprowadzi się do tego, by w zależności od różnych parametrów potrafił on podawać logiczną wartość **predykatu celu**: *Czekać*. Predykat ten będzie wyrażony przez drzewo decyzyjne, a poszczególne wartości predykatu będą umieszczone w liściach drzewa. Na początku, by przekształcić nasze rozważania na problem nauczania, musimy zdecydować się jakie parametry wejściowe będzie posiadał predykat *Czekać*:

1. *Alternatywa* : czy w okolicy jest inna restauracja.
2. *Bar* : czy restauracja ma wygodny bar, jako miejsce do oczekiwania.
3. *Pt/Sob* : prawdziwe, jeśli jest akuratnie piątek lub sobota.
4. *Głodny* : czy jesteśmy głodni.
5. *Ludzie* : ilość ludzi w restauracji (pełno, trochę, brak).
6. *Cena* : średnie ceny w restauracji ($, $$, $$$).
7. *Deszcz* : czy pada akuratnie deszcz.
8. *Rezerwacja* : czy mamy rezerwację.
9. *Typ* : rodzaj restauracji (Chińska, KFC, McDonald's, Pizzeria)
10. *ŚrCzas* : przeciętny czas oczekiwania klienta (0-10 minut, 10-30, 30-60, >60).

Oryginalne drzewo decyzyjne reprezentujące ten problem zostało skonstruowane tak, jak przedstawia to rysunek nr 27



Rysunek 27 Oryginalne drzewo decyzyjne dla problemu "czekania na stolik"

Warto zauważyć, że powyższe drzewo decyzyjne nie korzysta w ogóle z parametrów Typ i Cena, gdyż na podstawie danych, które agent zebrał, parametry te nie mają wpływu na wartość predykatu *Czekaj*. Teoretycznie każde drzewo może być wyrażone przez koniunkcję poszczególnych implikacji, z których każda reprezentuje ścieżkę drzewa od korzenia, aż do liścia „Tak". Przykładowo, ścieżka dla restauracji, w której jest pełno ludzi, średni czas oczekiwania waha się od 0-10 minut, podczas gdy agent nie jest głodny ma postać:

$$\forall r \; Ludzie(r,Pełno) \wedge ŚrCzas(r,0\text{-}10) \wedge Głodny(r,Nie) \Rightarrow Czekaj(r)$$

Możliwości stosowania drzew decyzyjnych

Jeśli drzewu decyzyjnemu odpowiada zbiór zdań – implikacji, to naturalnym pytaniem jest, czy drzewa decyzyjne mogą reprezentować dowolny taki zbiór. Odpowiedź brzmi „nie", gdyż drzewa decyzyjne mogą opisywać tylko pojedynczy obiekt. Oznacza to, że drzewa decyzyjne funkcjonują tylko na języku zdań, a każdy parametr jest pojedynczym zdaniem. Nie możemy używać

drzew decyzyjnych do analizy problemów, które odnoszą się do kilku różnych obiektów, np. gdy pytamy się, czy w okolicy znajduje się restauracja, która szybciej podaje posiłki:

$\exists r_2 \; Blisko(r,r_2) \wedge \acute{S}rCzas(r,c_1) \wedge \acute{S}rCzas(r_2,c_2) \wedge Szybciej(c_2,c_1)$

Oczywiście moglibyśmy dodać kolejny binarny parametr *SzyszaRestWPobliżu*, ale dodawanie *wszystkich* tego typu parametrów mijało by się z celem.

Drzewa decyzyjne mogą w pełni wyrażać klasę języków zdaniowych, oznacza to, że każda funkcja binarna znajdzie swoją reprezentację w postaci drzewa logicznego. Konstrukcja takiego drzewa decyzyjnego nie jest trudna. Wystarczy, że każdy wiersz (w którym są kolejne wartości wszystkich parametrów) w tablicy wartościowania funkcji binarnej przedstawimy jako koniunkcję parametrów, która reprezentować będzie ścieżkę w drzewie. To nie będzie koniecznie najlepszy sposób reprezentacji funkcji binarnej, ponieważ tablica wartościowania funkcji jest bardzo duża (rośnie ekspotencjalnie wraz z liniowym wzrostem ilości parametrów). Gdy przykładowo n - parametrowa funkcja binarna daje zawsze 0, wtedy można skonstruować drzewo decyzyjne , które ma 1 zamiast 2^n gałęzi.

Są funkcje, dla których użycie drzew binarnych byłoby wyjątkowo niekorzystne. Przykładowo funkcja parzystości, która daje wynik 1, wtedy i tylko wtedy, gdy parzysta ilość parametrów ma wartość 1. Podobnie trudno jest zbudować drzewo decyzyjne dla funkcji mniejszości, która zwraca wartość 1, jeśli co najwyżej połowa jej parametrów ma wartość 1.

Reasumując, drzewa decyzyjne są dla niektórych funkcji odpowiednie, dla niektórych zaś nie. Okazuje się, że nie istnieje jedna reprezentacja, która byłaby efektywna dla wszystkich funkcji logicznych. Uzasadnienie jest stosunkowo proste. Wystarczy rozpatrzyć n – parametrowe funkcje binarne. Tablica wartościowania posiada więc 2^n wierszy. Każdy taki wiersz reprezentuje obiekt złożony z wartości kolejnych n elementów. Dziedzina funkcji posiada zatem 2_n obiektów, stąd jeśli rozpatrzymy ilość funkcji, które różnią się od siebie przynajmniej na jednym obiekcie dziedziny, to będzie ich razem 2^{2^n}. Taka więc jest liczba różnych funkcji n – parametrowych, w związku z czym nie może istnieć jeden, uniwersalny sposób efektywnej prezentacji wszystkich funkcji.

Indukowanie drzew decyzyjnych z przykładów

Mianem **wzorca** będziemy teraz określać obiekt zawierający wartości wszystkich parametrów predykatu *Czekać*, oraz samą decyzję (Tak/Nie). Na początku dokonamy klasyfikacji wzorców: te, które dadzą jako wynik predykatu *Czekaj* prawdę (*Tak*) będziemy nazywali **wzorcami pozytywnymi**, a te, które dadzą fałsz (*Nie*) będą **wzorcami negatywnymi**.

W tabeli, którą za chwile przedstawimy znajduje się zbiór wzorców określany mianem **zbioru treningowego**. Spośród wzorców X_1, ..., X_{12} wyróżniamy wzorce pozytywne, dla których predykat Czekaj daje wynik Tak (X_1, X_3, ...), oraz wzorce negatywne (X_2, X_5, ...).

Wzorzec	Parametry										Cel: Czekaj
	Alt	Bar	Pt/S	Gło	Lud	Cena	Deszcz	Rez	Typ	ŚrCzas	
X_1	*Tak*	*Nie*	*Nie*	*Tak*	*Trochę*	*$$$*	*Nie*	*Tak*	*Chiń*	*0-10*	Tak
X_2	*Tak*	*Nie*	*Nie*	*Tak*	*Pełno*	*$*	*Nie*	*Nie*	*McD*	*30-60*	Nie
X_3	*Nie*	*Tak*	*Nie*	*Nie*	*Trochę*	*$*	*Nie*	*Nie*	*Pizz*	*0-10*	Tak
X_4	*Tak*	*Nie*	*Tak*	*Tak*	*Pełno*	*$*	*Nie*	*Nie*	*McD*	*10-30*	Tak
X_5	*Tak*	*Nie*	*Tak*	*Nie*	*Pełno*	*$$$*	*Nie*	*Tak*	*Chiń*	*>60*	Nie
X_6	*Nie*	*Tak*	*Nie*	*Tak*	*Trochę*	*$$*	*Tak*	*Tak*	*KFC*	*0-10*	Tak
X_7	*Nie*	*Tak*	*Nie*	*Nie*	*Brak*	*$*	*Tak*	*Nie*	*Pizz*	*0-10*	Nie
X_8	*Nie*	*Nie*	*Nie*	*Tak*	*Trochę*	*$$*	*Tak*	*Tak*	*McD*	*0-10*	Tak
X_9	*Nie*	*Tak*	*Tak*	*Nie*	*Pełno*	*$*	*Tak*	*Nie*	*Pizz*	*>60*	Nie
X_{10}	*Tak*	*Tak*	*Tak*	*Tak*	*Pełno*	*$$$*	*Nie*	*Tak*	*KFC*	*10-30*	Nie
X_{11}	*Nie*	*Nie*	*Nie*	*Nie*	*Brak*	*$*	*Nie*	*Nie*	*McD*	*0-10*	Nie
X_{12}	*Tak*	*Tak*	*Tak*	*Tak*	*Pełno*	*$*	*Nie*	*Nie*	*Pizz*	*30-60*	Tak

Patrząc na powyższą tabelę możemy dojść do wniosku, że znalezienie drzewa decyzyjnego jest procesem bardzo skomplikowanym, jednak w rzeczywistości okazuje się, że jest to proces trywialny.

Zauważmy, że możemy przecież potraktować każdy wzorzec osobno i zbudować dla niego oddzielną ścieżkę drzewa decyzyjnego. Kolejne atrybuty są wtedy węzłami drzewa; chcąc przetestować nauczonego agenta na danym wzorcu, rezultat będzie znaleziony poprzez podążanie tą samą ścieżką, co w procesie nauczania. Niestety, takie drzewo decyzyjne będzie miało mało do powiedzenia o innych klasach!

Problemem takiego drzewa jest fakt, że służy ono tylko do zapamiętania informacji przekazanej przez wzorce. Nie rozszerza ono swoich decyzji na różne od wzorców elementy przestrzeni (różne zestawienia parametrów wejściowych), dlatego nie możemy tu mówić o samodzielnym podejmowaniu decyzji.

Rozszerzanie klasy, dla której agent jest w stanie podejmować decyzje nie powinno więc skupiać się tylko na tym, by drzewo decyzyjne było jak najbardziej rozbudowane w celu podejmowania dobrych decyzji wzorcach. Przeciwnie, chodzi o to, żeby drzewo działało na wzorcach, ale by było jak

najmniejsze (najprostsze), by potrafiło decydować o możliwie największej liczbie obiektów z otoczenia, w którym funkcjonuje. Przenosząc to na bardziej formalny język, chodzi o znalezienie jak najogólniejszej hipotezy, dającej na wzorcach poprawne rezultaty. Tu pojawia się problem nauczania agenta, ponieważ trudniej jest znaleźć prostą hipotezę, niż jak widzieliśmy wcześniej, znaleźć skomplikowaną hipotezę, która nie wnosi niczego dla elementu nauczania.

Znajdowanie hipotezy najmniejszej jest zadaniem trudnym do wykonania, można jednak próbować znaleźć hipotezy „w miarę" proste. Ideą stojącą za algorytmem DRZ_DECYZ_NAUCZANIE jest, by testować najpierw parametry, które wprowadzają największy podział pomiędzy wzorcami. Dzięki takiej metodzie spodziewamy się otrzymać poprawną klasyfikację wzorców wykonując małą liczbę testów, przez co długość ścieżek drzewa będzie możliwie mała i ostatecznie drzewo decyzyjne (czyli i sama hipoteza końcowa) będzie małe.

Na rysunku nr 28 pokazane jest początkowe działanie algorytmu. Na starcie dysponujemy 12 wzorcami, które klasyfikujemy na pozytywne i negatywne. Następnie wybieramy parametr, który będzie użyty jako pierwszy test (będący w pierwszym węźle drzewa, czyli w korzeniu). Rysunek nr 28(a) pokazuje, że atrybut *Ludzie* jest dosyć ważny, ponieważ dla wartości „*Brak*" i „*Trochę*" możemy jednoznacznie określić wartość logiczną wzorców (jeśli wartością jest „Pełno" potrzebujemy dodatkowych testów). Na rysunku nr 28(b) widzimy, że Typ nie jest dobrym parametrem testowym, ponieważ po jego zastosowaniu otrzymamy 4 zbiory wzorców, przy czym w każdym zbiorze jest tyle samo wzorców pozytywnych, co negatywnych. By dojść do liści musielibyśmy do każdego z tych zbiorów zastosować co najmniej jeden inny parametr, w rezultacie czego prawie na pewno otrzymalibyśmy drzewo większe niż drzewo, którego korzeniem byłby parametr *Ludzie*.

Musimy przeanalizować każdy parametr by w końcu wybrać ten, który jest w danym momencie najważniejszy. W dalszej części tej sekcji wyjaśnimy, w jaki sposób algorytmicznie sprawdzać wagę operatora. Wróćmy więc do sytuacji, gdy jako korzeń został wybrany parametr *Ludzie*.

W tym momencie następuje rekursywne wywołanie algorytmu dla poddrzew, których korzeniami są dzieci korzenia Ludzie. Każde z tych poddrzew posiada swój zestaw wzorców, będący podzbiorem zbioru wyjściowego (żaden wzorzec nie występuje w 2 różnych poddrzewach). W tym momencie dla każdego poddrzewa stosujemy poniższe zasady:

1. Jeśli ma ono zarówno pozytywne jak i negatywne wzorce, wówczas wybierz najważniejszy parametr, by je rozdzielić. Na rysunku nr 28(c) pokazany jest wybór parametru Głodny.

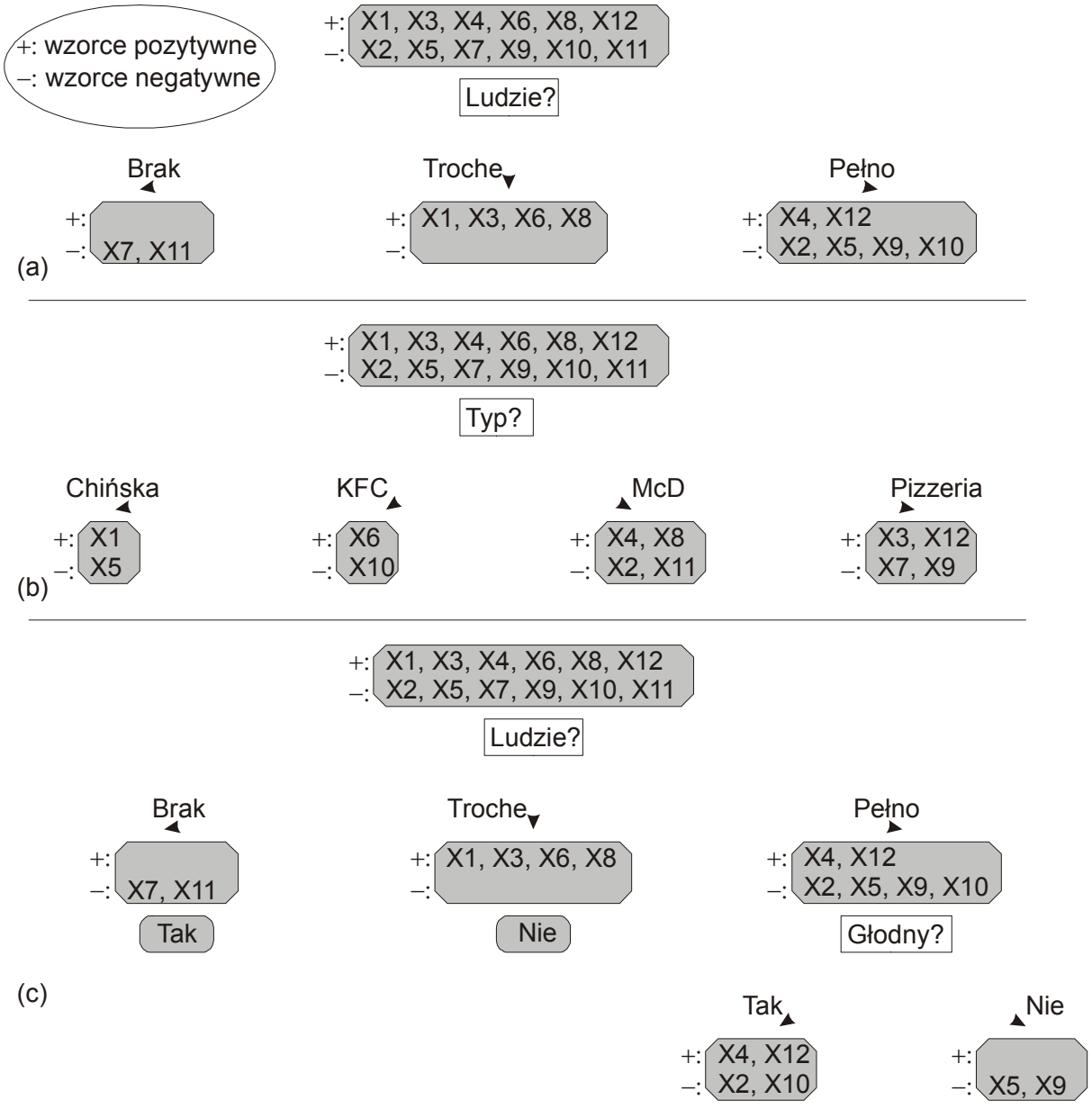
2. Jeśli wszystkie jego wzorce są pozytywne (lub wszystkie negatywne), wtedy możemy odpowiedzieć Tak lub Nie (wartość funkcji jest stałą dla całego poddrzewa).
3. Jeśli nie ma żadnych wzorców, oznacza to, że wzorzec o takich parametrach (określonych przez ścieżkę od korzenia drzewa głównego do poddrzewa) nie został zaobserwowany, a węzeł otrzymuje dowolną wartość od swojego rodzica (wartość, która występuje częściej).
4. Jeśli nie ma już parametrów, a nadal występują wzorce pozytywne i negatywne, oznacza to problem. Innymi słowy wzorce negatywne i pozytywne w myśl parametrów środowiska mają taki sam opis, skąd możemy wnioskować, że wystąpił błąd w danych (**hałas** w danych). Zdarza się tak również, gdy parametry nie dostarczają wystarczającej ilości informacji by opisać zadaną sytuację, lub gdy dziedzina problemu (zadana funkcja) jest niedeterministyczna. Sposobem na ten problem jest przypisanie takiej wartości, którą posiada większość wzorców w danym węźle drzewa.

Sam algorytm wygląda następująco:

function DRZ_DECYZ_NAUCZANIE(*wzorce,parametry,*$w_{domyślna}$)
returns *drzewo_decyzyjne*
inputs: *wzorce* – zbiór przekazywanych dl algorytmu wzorców
parametry – zbiór parametrów
$w_{domyślna}$ – domyślna wartość funkcji celu.

if *wzorce* = ∅ **then return** $w_{domyślna}$
else if wszystkie w*zorce* dają jedną wartość **then return** WARTOŚĆ*(wzorce)*
else if *parametry* = ∅ **then return** NAJCZĘSTSZA_WARTOŚĆ(*wzorce*)
else
$P_{najlepszy}$ ← WYBIERZ_PARAMETR(*parametry,wzorce*)
Drzewo ← nowe drzewo z korzeniem $P_{najlepszy}$
for each wartość w_i parametru $P_{najlepszy}$ **do**
$wzorce_i$ ←{elementy zbioru *wzorce* przy $P_{najlepszy} = w_i$ }
PD ←DRZ_DECYZ_NAUCZANIE($wzorce_i$, *parametry* - $P_{najlepszy}$,
NAJCZĘSTSZA_WARTOŚĆ(*wzorce*))
dodaj do *Drzewo* gałąź z wartością w_i i poddrzewem *PD*.
end
return *Drzewo*

Kontynuując działanie algorytmu dostajemy ostatecznie drzewo pokazane na rysunku nr 29. Drzewo to znacząco różni się od drzewa z rysunku nr 27 ,mimo iż było ono generowane dla takiego samego zestawu wzorców.



Rysunek 28 Działanie algorytmu tworzącego drzewo decyzyjne

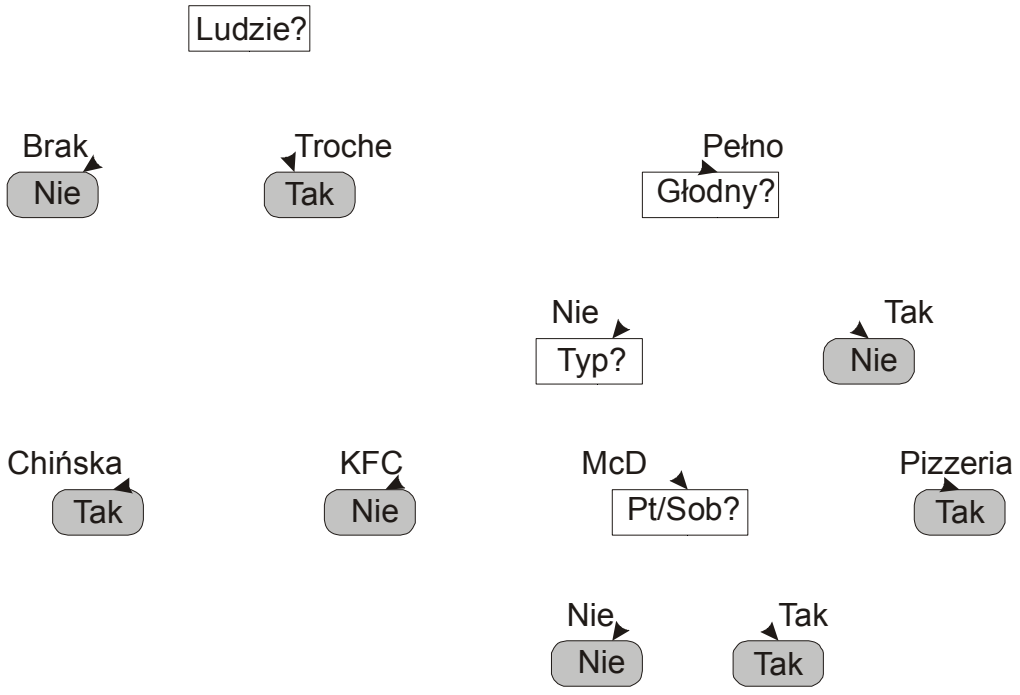
Można byłoby wywnioskować, że algorytm nauczania niezbyt radzi sobie z powierzonym mu zadaniem, skoro drzewo przez niego stworzone tak bardzo odbiega od oryginału. W rzeczywistości jest jednak inaczej. Znaleziona hipoteza sprawdza się dla wszystkich przekazanych algorytmowi wzorców i najlepiej jak to jest możliwe przybliża pierwotną funkcję (w tym wypadku oryginalne

drzewo). Możemy bez obaw przyznać, że hipoteza jest także zdecydowanie prostsza niż drzewo oryginalne. Algorytm nauczania nie ma żadnych powodów, by uwzględniać w końcowym drzewie testy *Deszcz* i *Rezerwacja*, ponieważ może on poprawnie sklasyfikować wszystkie wzorce bez tych parametrów.

Algorytm wykrył też interesującą nieregularność w danych wejściowych (Syty klient będzie czekał w pełnej restauracji McDonald's tylko w Piątki i Soboty).

Wraz ze wzrostem ilości wzorców ostateczne drzewo będzie coraz bardziej przypominało drzewo oryginalne. Drzewo na rysunku nr 29 jest jednak narażone na pewien błąd. Algorytm nie otrzymał żadnego wzorca, w którym *Czas Oczekiwania* jest 0-10, a restauracja jest *Pełna*. Weźmy głodnego klienta, który wchodzi do pełnej restauracji, w której będzie czekał maksymalnie 10 minut. Jest bardzo prawdopodobne, że klient taki zdecyduje się poczekać. Drzewo wygenerowane przez algorytm mówi jednak, że nie powinien on czekać.

To nasuwa pytanie: czy algorytm generuje konsekwentne, ale niepoprawne rozwiązania? Odpowiedź na podstawie eksperymentów brzmi: Im większa jest ilość wzorców przekazanych algorytmowi, tym większe jest prawdopodobieństwo, że wygenerowane drzewo będzie poprawne.



Rysunek 29 Drzewo decyzyjne utworzone na podstawie 12 wzorców

7.3 Korzystanie z teorii informacji

Ta sekcja poświęcona jest wyłącznie problemowi znalezienia najlepszego parametru, według którego algorytm dokonuje podziału zbioru wzorców. Omówimy tu na jakiej zasadzie działa procedura WYBIERZ_PARAMETR wykorzystana w algorytmie budowy drzewa decyzyjnego.

Potrzebna nam będzie funkcja, która na wejściu dostaje parametr, a na wyjściu podaje jego wartość. Docelowo więc bardziej ważne parametry będą miały większą wartość od mniej ważnych. Posłużymy się Teorią Informacji podaną przez Shannon'a w 1949 roku.

Dla zobrazowania zagadnienia przedstawimy prosty przykład. Załóżmy, że prezenter pogody mówi dwa zdania:

1 – W Jerozolimie spadł śnieg.

2 – W Sztokholmie spadł śnieg.

Zastanówmy się, która informacja jest dla nas ważniejsza, za którą informację moglibyśmy więcej zapłacić. Bez wątpienia prawdopodobieństwo zdarzenia opisanego przez pierwsze zdanie jest mniejsze i to właśnie to zdanie wywołałoby sensację. Możemy więc napisać, że im wydarzenie jest mniej prawdopodobne, tym więcej niesie ze sobą informacji.

Teoria informacji bazuje na tej samej intuicji, jednak wartość niesionej informacji podawana jest w **bitach**. Jeden bit informacji wystarcza by na pytanie, o którym nic nie wiemy odpowiedzieć Tak lub Nie. W rzeczywistości, jeśli możliwe odpowiedzi o_i mają prawdopodobieństwa $P(o_i)$, wówczas ilość informacji I przekazanej przez odpowiedź wyraża się wzorem:

$$I(P(o_1), ..., P(o_n)) = \sum_{i=1}^{n} -P(o_i) \log_2 P(o_i)$$

Jeśli weźmiemy dla przykładu 1 rzut zrównoważoną monetą, to informacja o wyniku rzutu zajmuje 1 bit, gdyż:

$$I(0.5, 0.5) = -0.5 \log_2(0.5) - 0.5 \log_2(0.5) = 1 \; bit$$

Dla problemu drzew decyzyjnych, pytaniem na które należy odpowiedzieć jest: dla danego wzorca, jaka jest poprawna wartość funkcji? Poprawne drzewo decyzyjne na takie pytanie odpowiada. Prawdopodobieństwo, że dla wybranego z zestawu wzorca otrzymamy konkretną wartość funkcji można obliczyć przez odpowiednie proporcje. Przykładowo, gdy zbiór treningowy liczy n wzorców negatywnych i p pozytywnych, wówczas informacja zawarta w poprawnej odpowiedzi wyraża się wzorem:

$$I\left(\frac{p}{p+n}, \frac{n}{p+n}\right) = -\frac{p}{p+n}\log_2\frac{p}{p+n} - \frac{n}{p+n}\log_2\frac{n}{p+n}$$

Dla zbioru treningowego dla problemu „czekania w restauracji" jest p=n=6 , zatem odpowiedź na pytanie, czy wzorzec jest prawdziwy zajmuje 1 bit informacji.

Pojedynczy parametr A zwykle nie poda nam dużo informacji (wiemy na razie ile potrzebujemy informacji przed jego zastosowaniem). Możemy zmierzyć ilość przekazywanych przez niego informacji sprawdzając ile będziemy potrzebowali informacji po zastosowaniu tego parametru jako testu w drzewie decyzyjnym. Niech parametr A dzieli zbiór wzorców na podzbiory E_1,E_2, ..., E_k w zależności od *k* różnych wartości parametru A. Każdy podzbiór E_i posiada p_i wzorców pozytywnych i n_i wzorców negatywnych, dlatego idąc wzdłuż tej gałęzi będziemy potrzebowali dodatkowych $I(p_i/(p_i+n_i), n_i/(p_i+n_i))$ bitów informacji by sklasyfikować wzorzec w danym podzbiorze. Losowy wzorzec znajduje się w podzbiorze E_i z prawdopodobieństwem $(p_i+n_i)/(p+n)$, więc po przetestowaniu parametru A, będziemy potrzebowali jeszcze

$$Reszta(\mathrm{A}) = \sum_{i=1}^{k} \frac{p_i + n_i}{p + n} I\left(\frac{p_i}{p_i + n_i}, \frac{n_i}{p_i + n_i}\right)$$

bitów informacji by sklasyfikować wszystkie wzorce. Zysk informacji po zastosowaniu operatora A jest więc różnicą pomiędzy informacją przed jego użyciem i Resztą:

$$Zysk(\mathrm{A}) = I\left(\frac{p}{p+n}, \frac{n}{p+n}\right) - Reszta(\mathrm{A})$$

, a funkcja WYBIERZ_PARAMETR wybiera właśnie ten parametr, który ma największy zysk.

Patrząc na wybór parametrów *Ludzie* i *Typ* z rysunku nr 28 możemy obliczyć:

$Zysk(Ludzie) = 1 - [(2/12)\cdot I(0,1) + (4/12)\cdot I(1,0) + (6/12)\cdot I(2/6,4/6)] \approx 0.541$ bitów

$Zysk(Typ) = 1 - [(2/12)\cdot I(1/2,1/2) + (2/12)\cdot I(1/2,1/2) + (4/12)\cdot I(2/4,2/4) + (4/12)\cdot I(2/4,2/4)] = 0$ bitów

, co uzasadnia wybór parametru *Ludzie* na korzeń drzewa decyzyjnego.

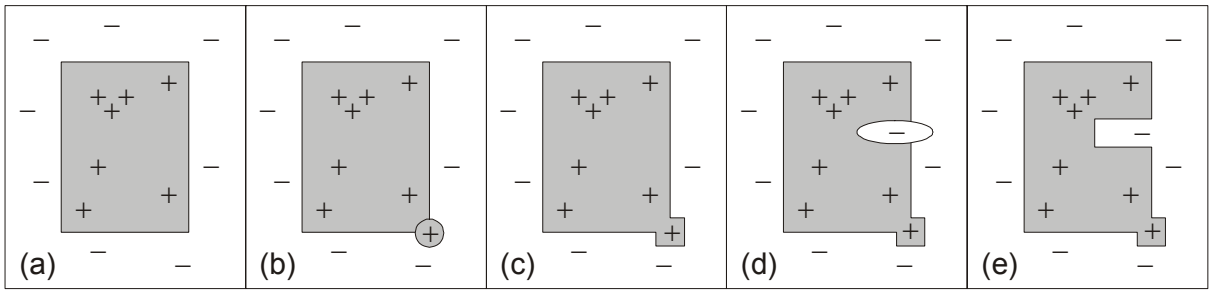
7.4 Inne metody szukania hipotez

Obok drzew decyzyjnych istnieją również inne sposoby znajdowania hipotez, jedną z nich jest Algorytm poszukiwania hipotezy **Aktualnie Najlepszej.**

Idea stojąca za tym algorytmem jest trywialna. Algorytm startuje z prostymi hipotezami zgodnymi z pierwszym wzorcem, a następnie, wraz z napływem nowych wzorców, hipotezy te modyfikuje, by zgadzały się dla wszystkich wzorców, które już nadeszły.

Załóżmy, że tak jak na Rysunku nr 30(a) posiadamy hipotezę H. Tak długo dopóki napływające wzorce są z nią zgodne nie ma potrzeby, by ją modyfikować, lecz okazuje się, że napływa nowy, pozytywny wzorzec X_n, który nie mieści się w zakresie działań hipotezy – Rysunek nr 30(b). Algorytm musi

zatem generalizować hipotezę H, by była zgodna z wzorcem X_n – Rysunek nr 30(c). Teraz z kolei przychodzi negatywny wzorzec X_m, który według istniejącej hipotezy H byłby sklasyfikowany jako negatywny – Rysunek nr 30(d). Istnieje więc konieczność specyfikacji hipotezy H, by nie zawierała ona w sobie wzorca X_m – Rysunek nr 30(e). Taką czynność kontynuujemy aż do przeanalizowania wszystkich wzorców.



Rysunek 30 Schemat działania algorytmu Aktualnie-Najlepszej hipotezy

Warto zauważyć, że za każdym razem, gdy generalizujemy lub specyfikujemy hipotezę H trzeba robić to ostrożnie, by nie naruszyć poprawności hipotezy wobec wzorców przeanalizowanych wcześniej.

Szkic algorytmu szukania Aktualnie Najlepszej Hipotezy przedstawiony jest poniżej:

```
function AKT_NAJLEP_NAUCZANIE(wzorce) returns hipoteza

H ← dowolna hipoteza zgodna z pierwszym wzorcem
    for each pozostały wzorzec wz w zbiorze wzorce do
        if wz jest błędnie prawdziwa dla H then
            H ← wybierz specyfikację H zgodną z wzorcami
        if wz jest błędnie fałszywa dla H then
            H ← wybierz generalizację H zgodną z wzorcami
        if H nie da się znaleźć specyfikacji/generalizacji than fail
    end
    return H
```

Warto zauważyć, że powyższy algorytm nie jest deterministyczny, ponieważ za każdym razem jest wiele możliwości specyfikacji/generalizacji hipotezy H – stąd algorytm niekoniecznie znajduje najprostszą hipotezę. Jeśli w danym etapie algorytmu nie będzie możliwa modyfikacja H, by hipoteza akceptowała wszystkie wzorce, algorytm musi się cofnąć do wcześniejszego miejsca i wybrać inną specyfikację/generalizację.

7.5 Zadania

1. Załóżmy, że mamy agenta, który chce wybrać się na wakacje. Chcemy go nauczyć wybierać dobre propozycje wakacyjne. Wybór agenta jest funkcją, której parametrami są własności oferty wakacyjnej; rezultatem tej funkcji jest decyzja Tak / Nie w zależności od tego, czy akceptujemy ofertę, czy nie. Parametrami są:

Cena: *Duża, Średnia, Niska*
Transport: *Auto, Autokar, Samolot*
Kraj: *Francja, Hiszpania, Grecja*
Hotel: **, ***, ****

Poniżej przedstawiony jest zbiór treningowy liczący 8 wzorców:

Wzorce	Parametry				**Wybieram?**
	Cena	Transport	Kraj	Hotel	
X1	*Duża*	*Autokar*	*Grecja*	****	**Tak**
X2	*Średnia*	*Auto*	*Grecja*	**	**Nie**
X3	*Średnia*	*Samolot*	*Francja*	***	**Tak**
X4	*Duża*	*Samolot*	*Hiszpania*	****	**Tak**
X5	*Niska*	*Auto*	*Hiszpania*	***	**Tak**
X6	*Niska*	*Auto*	*Hiszpania*	**	**Nie**
X7	*Niska*	*Samolot*	*Francja*	**	**Tak**
X8	*Duża*	*Samolot*	*Francja*	***	**Nie**

Wykorzystaj teorię informacji do zbudowania minimalnego, poprawnego drzewa decyzyjnego.

2. Dla takiego samego problemu jak w zadaniu poprzednim, znajdź optymalną hipotezę posługując się algorytmem znajdowania Aktualnie Najlepszej Hipotezy.

Literatura

1. Stuart Russell, Peter Norvig „Artificial Intelligence, a modern Approach" 1995 Prentice – Hall.
2. L. Ardissono , G. Boella , D. Sestero "Recognition of Preliminary Sentences in Dialogue Interpretation" Springer Verlag, 1995.
3. M. E. Stickel "An introduction to automated deduction", , Springer, 1987.
4. N. I. BadlerArtificial "Intelligence, Natural Language and Simulation for Human Animation State-of-the-Art in Computer Animation", Springer Verlag, 1989.
5. R. Pfeifer ,C. Scheier Introduction to "New Artificial Intelligence", AI Lab, Computer Science Department, University of Zurich, November 1995
6. J. M. Alliot, T. Schiex „Intelligence artificielle & informatique theorique", Cepadues-editions Toulousse, 1994
7. D. Kayser "Representation de connaissances" Hermes, Paris 1998
8. A. M. Turing. "Computing machinery and intelligence" Mind, 59 (1950), 433-460
9. Antoni Niederlinski „Systemy Eksperckie"
10. H. Kowalowski „Inżynieria Wiedzy" – Wydawnictwo Pracowni Komputerowej Jacka Skalmierskiego, Gliwice 2000