Politechnika Warszawska

Ośrodek Kształcenia Na Odległość

Metody Sztucznej Inteligencji (studia inżynierskie)

Autor: Włodzimierz Kasprzak

Opracowanie multimedialne:

Warszawa, grudzień 2015

Metody Sztucznej Inteligencji

Przedmiot: Metody Sztucznej Inteligencji – studia inżynierskie

Edycja: 3.0

Słowa kluczowe: podręcznik multimedialny, e-book, wnioskowanie, przeszukiwanie, uczenie

Autor: Włodzimierz Kasprzak <W.Kasprzak(at)elka.pw.edu.pl>
Opracowanie multimedialne:

Spis treści

Metody Sztucznej Inteligencji (studia inżynierskie) 1
Spis treści 3
1. Zadania Sztucznej Inteligencji 11
1.1. Wprowadzenie 11
1.1.1. Działać racjonalnie 11
1.1.2. Zadania i zastosowania Sztucznej Inteligencji 12
1.1.3. System agentowy 12
1.2. System z bazą wiedzy 14
1.2.1. Baza wiedzy 14
1.2.2. Podsystem sterowania 14
1.3. Język reprezentacji wiedzy 16
1.4. System logicznego wnioskowania 16
1.4.1. Wynikanie zdań 16
1.4.2. Wnioskowanie 17
1.4.3. System logiczny 18
1.5. Przeszukiwanie 18
1.6. Uczenie 19
1.7. Pytania 20
1.8. Zadania 20
2. Rachunek zdań 23
2.1. Składnia rachunku zdań 23
2.2. Semantyka rachunku zdań 23
2.3. Wnioskowanie 24
2.3.1. Wnioskowanie z tabelą prawdy 24
2.3.2. Twierdzenia o dedukcji 25
2.3.3. Przekształcenia równoważne 25
2.3.4. Reguły wnioskowania 26
2.4. Postacie normalne 27
2.4.1. Klauzula Horna i reguła odrywania 27
2.4.2. Postać normalna CNF i reguła rezolucji 27
2.5. Wnioskowanie poprzez rezolucję (dowód nie wprost) 28
2.6. Wnioskowanie z regułą odrywania (generacja lub dowód wprost) 29
2.6.1. Wnioskowanie w przód 29
2.6.2. Wnioskowanie wstecz 31
2.7. Własności zmienne w czasie 33
2.8. Pytania 33
2.9. Zadania 33

3. Logika predykatów – reprezentacja wiedzy 36
3.1. Składnia języka logiki pierwszego rzędu 36
3.1.1. Kategorie symboli języka 36
3.1.2. Wyrażenia – termy i formuły 36
3.1.3. Kwantyfikatory 37
3.2. Semantyka języka predykatów 38
3.2.1. Model i wartościowanie zmiennych 38
3.2.2. Rodzaje formuł 39
3.3. Przekształcanie formuł 39
3.3.1. Postać predykatowa 39
3.3.2. Podstawienie pod zmienne 40
3.3.3. Uzgadnianie zmiennych i unifikacja formuł 40
3.3.4. Standaryzacja rozłączna 41
3.3.5. Eliminacja kwantyfikatorów 42
3.4. Rachunek sytuacji w logice predykatów 43
3.4.1. Przykład 43
3.4.2. Wymagane elementy 43
3.4.3. Aksjomaty efektów akcji 44
3.4.4. Aksjomaty tła 44
3.4.5. Aksjomaty następstwa stanów 44
3.5. Pytania 44
3.6. Zadania 44
4. Wnioskowanie w logice predykatów 46
4.1. Twierdzenia o dedukcji 46
4.2. Postaci normalne formuł i reguły wnioskowania 46
4.2.1. Klauzula Horna i uogólniona reguła odrywania 46
4.2.2. Postać normalna CNF i reguła rezolucji 47
4.3. Procedury wnioskowania 48
4.3.1. Wnioskowanie poprzez rezolucję 48
4.3.2. Wnioskowanie w przód 49
4.3.3. Wnioskowanie wstecz 49
4.4. System logicznego wnioskowania 50
4.4.1. Przykłady 50
4.4.2. Zadania systemu 50
4.4.3. Implementacja 51
4.5. Realizacja celu jako rozwiązanie problemu 51
4.5.1. Przykład 51
4.5.2. Techniki realizacji celu 51
4.5.3. Wnioskowanie jako realizacja celu 52

4.6. Pytania 52
4.7. Zadania 53
5. System ekspertowy 55
5.1. Inżynieria wiedzy 55
5.1.1. Zasady wyznaczania bazy wiedzy 55
5.1.2. Przykład 55
5.2. Ontologia języka 57
5.3. System ekspertowy 61
5.4. PROLOG 62
5.5. System regułowy 63
5.5.1. Faza dopasowania 64
5.5.2. Faza wyboru akcji 64
5.6. Sieci semantyczne (ramy) 65
5.6.1. Sieć semantyczna 65
5.6.2. Logika a sieć semantyczna 65
5.6.3. Wyjątki 66
5.6.4. Logika niemonotoniczna 66
5.7. Pytania 66
5.8. Zadania 67
6. Przeszukiwanie przestrzeni stanów 71
6.1. Przestrzeń stanów 71
6.1.1. Przykład 71
6.1.2. Problem przeszukiwania 71
6.1.3. Przeszukiwanie drzewa 73
6.1.4. „Przeszukiwanie grafu” 75
6.2. Strategie ślepego przeszukiwania 76
6.2.1. Kryteria oceny strategii przeszukiwania 76
6.2.2. Przeszukiwanie wszerz – BFS 76
6.2.3. Przeszukiwanie w głąb – DFS 77
6.2.4. Przeszukiwanie z jednolitą funkcją kosztu - UCS 79
6.2.5. Przeszukiwanie z ograniczoną głębokością – DLS 80
6.2.6. Przeszukiwanie z iteracyjnym pogłębianiem – IDS 80
6.2.7. Porównanie strategii przeszukiwania 81
6.3. Pytania 82
6.4. Zadania 82
7. Przeszukiwanie poinformowane 84
7.1. Strategie „najpierw najlepszy” (*best first*) 85
7.2. Strategia zachłanna („najbliższy celowi najpierw”) 86
7.3. Przeszukiwanie A* 89

7.3.1. Implementacja strategii A* ... 91
7.3.2. Dopuszczalna heurystyka ... 91
7.3.3. Spójna heurystyka ... 92
7.4. Wyznaczanie składowej heurystycznej ... 92
7.4.1. Dominująca heurystyka ... 92
7.4.2. Generowanie heurystyk metodą „złagodzonego problemu” ... 93
7.5. Pytania ... 94
7.6. Zadania ... 94
8. Losowe i lokalne przeszukiwanie ... 96
8.1. Losowość w przeszukiwaniu ... 96
8.1.1. Algorytmy losowego próbkowania ... 96
8.1.2. Algorytm błądzenia przypadkowego ... 97
8.2. Przeszukiwanie lokalne poinformowane ... 97
8.2.1. Przeszukiwanie przez „wspinanie” ... 98
8.2.2. Przeszukiwanie lokalne w dziedzinie ciągłej ... 100
8.3. Symulowane wyżarzanie ... 100
8.4. Algorytmy genetyczne ... 101
8.5. Pytania ... 102
8.6. Zadania ... 103
9. Gry dwuosobowe ... 104
9.1. Drzewo gry typu „Mini-max” ... 104
9.2. Strategia Mini-max ... 105
9.3. Cięcia alfa-beta ... 106
9.3.1. Zasada przycinania drzewa ... 106
9.3.2. Implementacja strategii „Mini-Max z cięciami alfa-beta” ... 108
9.3.3. Własności strategii „cięć alfa-beta” ... 109
9.4. Heurystyczna ocena – „obcięty Mini-Max” ... 110
9.4.1. Zasada modyfikacji „Mini-Max”-u w praktyce ... 110
9.4.2. Implementacja obciętego „Mini-Max”-u ... 110
9.5. Pytania ... 111
9.6. Zadania ... 111
10. Uczenie się na podstawie obserwacji - indukcja ... 114
10.1. Uczenie się przez indukcję ... 114
10.1.1. Cel uczenia ... 114
10.1.2. Pojęcia ... 114
10.1.3. Zbiór treningowy (próbki uczące) ... 114
10.1.4. Sposoby uczenia ... 115
10.1.5. Uczenie się pojęć ... 115
10.1.6. Tworzenie pojęć ... 115

10.1.7. Uczenie się aproksymacji ... 115
10.2. Tryb podawania próbek ... 115
10.2.1. Tryby podawania próbek uczących ... 116
10.2.2. Model ograniczania pomyłek ... 117
10.3. Uczenie się pojęć ... 117
10.3.1. Własności hipotezy ... 117
10.3.2. Przestrzeń wersji ... 118
10.3.3. Algorytm „Find S” ... 119
10.3.4. Algorytm Eliminacji Kandydatów ... 120
10.4. Pytania ... 122
10.5. Zadania ... 122
11. Uczenie się klasyfikacji ... 123
11.1. Zadanie klasyfikacji ... 123
11.2. Drzewa decyzyjne ... 123
11.2.1. Siła wyrazu drzew decyzyjnych ... 125
11.2.2. Uczenie się drzewa decyzyjnego ... 125
11.2.3. Kryteria wyboru testów ... 126
11.3. Klasyfikator numeryczny ... 128
11.3.1. Definicja problemu ... 128
11.3.2. Podstawowe rodzaje klasyfikatorów numerycznych ... 129
11.4. Klasyfikator według funkcji potencjału ... 130
11.4.1. Liniowe funkcje potencjału ... 130
11.4.2. Uczenie się parametrów klasyfikatora ... 131
11.5. Klasyfikator SVM ... 132
11.5.1. SVM dla liniowo separowalnego zbioru próbek ... 133
11.5.2. Liniowa SVM dla liniowo nieseparowalnych próbek ... 135
11.5.3. Nieliniowe maszyny SVM ... 135
11.6. Pytania ... 136
11.7. Zadania ... 137
12. Uczenie się aproksymacji ... 138
12.1. Zadanie aproksymacji funkcji ... 138
12.2. Metodyka uczenia się aproksymacji funkcji ... 138
12.3. Funkcja parametryczna ... 140
12.3.1. Funkcja liniowa ... 140
12.3.2. Reguła poszukiwania wzdłuż ujemnego gradientu ... 141
12.4. Model pamięciowy aproksymacji funkcji ... 142
12.5. Pytania ... 143
12.6. Zadania ... 143
13. Uczenie sieci neuronowej MLP ... 146

13.1. Model neuronu ... 146
13.1.1. Struktura neuronu ... 146
13.1.2. Funkcja aktywacji neuronu ... 146
13.2. Wielowarstwowy perceptron ... 147
13.2.1. Struktura sieci MLP ... 147
13.2.2. Parametry sieci MLP ... 148
13.3. Uczenie sieci MLP ... 150
13.3.1. Reguła modyfikacji wag ... 150
13.3.2. Warstwa wyjściowa ... 150
13.3.3. Warstwy ukryte ... 150
13.4. Pytania ... 151
13.5. Zadania ... 151
Bibliografia ... 153

Część I: Inżynieria wiedzy (reprezentacja i wnioskowanie)

1. Zadania Sztucznej Inteligencji

1.1. Wprowadzenie

1.1.1. Działać racjonalnie

Próbując zdefiniować czym jest **Sztuczna Inteligencja** przyjęło się wyróżniać cztery podejścia do tego zagadnienia, określane alternatywnie jako [4]:

1. Myśleć jak człowiek,
2. Myśleć racjonalnie,
3. Działać jak człowiek,
4. Działać racjonalnie.

W latach 1960-ych pojawiła się dziedzina wiedzy zwana „kognitywistyką". Z początku ograniczona była do „psychologii poznawczej", czyli psychologii przetwarzania informacji. Kognitywistyka tworzy naukowe teorie aktywności mózgu. Powstaje pytanie, jak te teorie należy weryfikować? Możliwe są zasadniczo dwa sposoby:

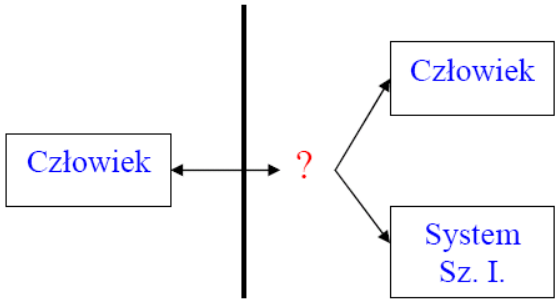
1. Przewidzieć model i testować zachowania osób (tzw. podejście „top-down"), co czyni psychologia poznawcza,
2. Bezpośrednia identyfikacja modelu na podstawie danych neurologicznych (tzw. podejście „bottom-up") – jest to przedmiotem „neuro-kognitywistyki".

Oba podejścia („cognitive science" i „cognitive neuroscience") są obecnie oddzielone od Sztucznej Inteligencji.

Podejście zwane „myśleć racjonalnie" bazuje na badaniach logiki myślenia. Już w starożytności Arystoteles badał to, jakie argumenty i procesy myślowe *są prawidłowe*, a greckie szkoły logiczne wprowadziły *zapis* i *reguły wyprowadzania* myśli. Problemy tego podejścia do inteligencji:

1. Nie każde inteligentne zachowanie jest wynikiem logicznego wyprowadzenia.
2. Czy potrafimy odpowiedzieć na pytania" „Jaki jest cel myślenia?", „Jakie myśli są prawidłowe?".

Podejście zwane „działać jak człowiek" sięga słynnego Testu Turinga (1950). Alan Turing tłumaczył pytanie „czy maszyna potrafi myśleć?" na to „czy maszyna działa inteligentnie (tak jak człowiek)?" Test Turinga to test inteligentnego zachowania się maszyny – jest to próba imitowania człowieka przez komputer (rys. 1.1).



Rys. 1.1: Idea „Testu Turinga"

Pionierskie prace Turinga zaowocowały zdefiniowaniem głównych zadań tego typowego podejścia do „Sztucznej Inteligencji": reprezentacja wiedzy, wnioskowanie, przeszukiwanie, uczenie.

Nasze podejście w trakcie tego kursu to: „**działać racjonalnie**”. Racjonalne działać oznacza „wykonywać właściwą akcję”. Właściwa akcja to taka, która maksymalizuje osiągnięcie celu przy zadanej informacji. Niekoniecznie wymaga to myślenia, np. może to być też nieświadomy refleks, ale myślenie powinno być na usługach racjonalnego działania. Takie podejście wymaga powiększenia zadań Sztucznej Inteligencji zagadnienia percepcji środowiska i wykonywania akcji. Zalety podejścia „działać racjonalnie” w porównaniu do poprzednich to:

- Pojęcie „działania” jest ogólniejsze od pojęcia „myślenia”.
- Pojęcie „racjonalny” jest lepiej określone niż „ludzki”.

1.1.2. Zadania i zastosowania Sztucznej Inteligencji

Główne **zadania** „Sztucznej Inteligencji”, w rozumieniu „jak działać racjonalnie”, to:

1. reprezentacja wiedzy i wnioskowanie,
2. rozwiązywanie problemów (przeszukiwanie przestrzeni stanów, planowanie działań),
3. uczenie (nabywanie wiedzy).

Główne zastosowania metod **sztucznej inteligencji** to jak dotąd dwa rodzaje systemów obliczeniowych:

1. **Systemy ekspertowe,** realizujące podejście „myśleć racjonalnie”, wspomagające człowieka w problemach logistycznych, diagnostyce medycznej, reagowaniu w nieoczekiwanych sytuacjach lub podejmowaniu decyzji biznesowych;
2. **Systemy agentowe**, realizujące podejście ”działać racjonalnie”, mają posiadać wyższy poziom autonomii, posiadając zdolności percepcji, wykonywania akcji i uczenia się.

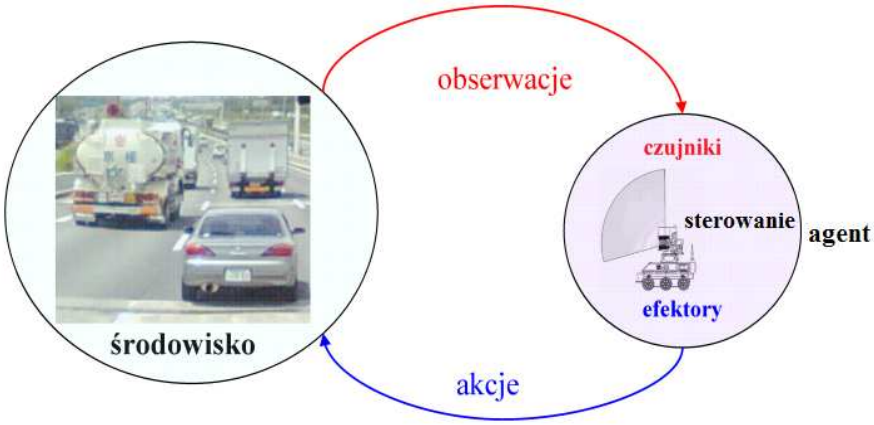
1.1.3. System agentowy

Agentem jest każdy obiekt (jednostka), który obserwuje (odbiera) swoje otoczenie dzięki czujnikom (sensorom) i oddziałuje na to środowisko przy pomocy „efektorów” (wykonuje akcje). Formalnie biorąc, działanie agenta opisuje funkcja z dziedziny historii obserwacji (percepcji) w wykonywane akcje (rys. 1.2):

$$[f\colon P^* \rightarrow A]$$

Ludzki agent posiada: oczy, uszy i inne organy pełniące role czujników oraz ręce, nogi, usta i inne części ciała będące efektorami. Agent upostaciowiony (robot, maszyna) posiada: kamery, czujniki podczerwieni, skanery laserowe, itp., będące czujnikami oraz różne silniki, napędy i manipulatory będące efektorami. Agent nieupostaciowiony (program komputerowy, *softbot*) to odpowiedni program wykonujący się na fizycznym komputerze (o określonej architekturze) i realizujący funkcję *f* agenta.

Głównym wyznacznikiem przy projektowaniu agenta będzie znalezienie najbardziej efektywnego działania w zależności od charakteru środowiska i rodzaju zadania. W przypadku agenta programowego wystąpi jeszcze problem ograniczenia zasobów komputera i wtedy naszym celem będzie: zaprojektowanie najlepszego programu z możliwych przy zadanych zasobach maszynowych.



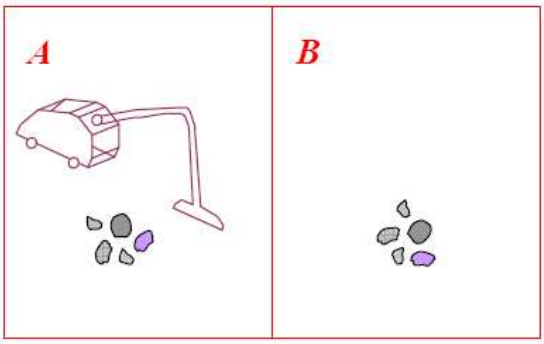
Rys. 1.2: Elementy systemu agentowego

Przykład. „Agent odkurzający” (rys. 1.3)

Percepcja tego agenta to 2-elementowy wektor: [położenie w kratce, stan czystości kratki].

Np. możliwe obserwacje to: [*A, kurz*], [*B, czysto*].

Załóżmy, że agent może wykonywać 4 akcje: { *Lewo, Prawo, Odkurzaj, NicNieRób* }. Program prostego agenta „odkurzającego” przedstawia Tablica 1-1.



Rys. 1.3: „Agent odkurzający” i jego środowisko

Tabl. 1-1. Funkcja prostego agenta-odkurzającego

```
function ProstyAgentOdkurzajacy( [lokalizacja, czystosc]) returns Akcja
{
  if (czystosc == kurz)  return Odkurzaj;
  else          if (lokalizacja == A)  return Prawo;
                else if (lokalizacja == B) return Lewo;
}
```

Zauważmy, że działanie (funkcja) powyższego agenta ma **charakter reaktywny** - agent nie posiada pamięci o przeszłych obserwacjach. Tym samym nie będzie on wiedział, kiedy się zatrzymać. Jeśli środowisko jest deterministyczne, to należałoby zatrzymać agenta wtedy, gdy w obu kratkach będzie czysto. To tego potrzebne jest jednak zapamiętanie przynajmniej jeszcze przedostatniej obserwacji. W ogólności rozpatruje się złożone funkcje agentowe, zależne od sekwencji obserwacji – system **pamięta** przeszłe obserwacje i akcje agenta w sposób bezpośredni lub przetworzony (np. w postaci wiedzy o stanach środowiska).

1.2. System z bazą wiedzy

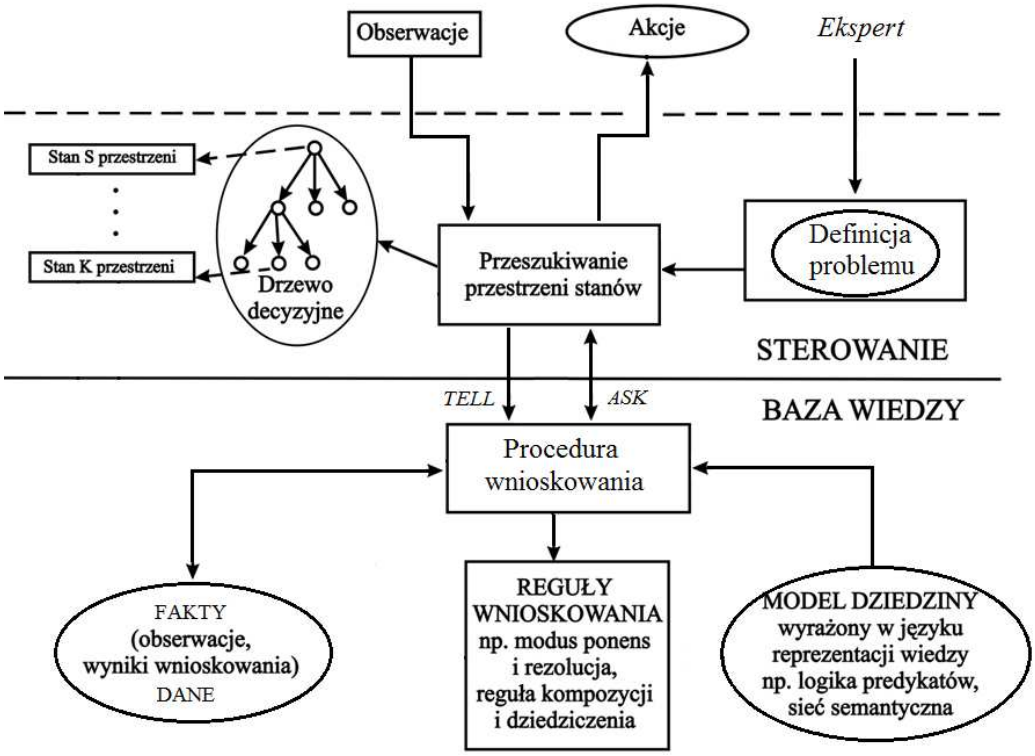
Implementacja informatyczna systemu ekspertowego lub agentowego następuje w oparciu o technologię **systemu z bazą wiedzy** (rys. 1.3).

1.2.1. Baza wiedzy

Głównym elementem systemu z bazą wiedzy jest **baza wiedzy** (ang. „*knowledge base*", KB), która zawiera zbiór **aksjomatów i faktów** o modelowanym świecie, wyrażony w języku reprezentacji wiedzy, oraz odpowiednie dla tego języka **reguły wnioskowania**.

Z punktu widzenia logiki matematycznej aksjomaty i fakty mają postać formuł zapisanych w języku reprezentacji wiedzy a reguły wnioskowania są **tautologiami** tego języka. Tautologia jest formułą zawsze prawdziwą w danym języku. Prawdziwość aksjomatów ograniczona jest do konkretnej dziedziny zastosowania systemu, natomiast prawdziwość formuł wyrażających fakty zależy od obserwacji aktualnego środowiska, od tego czy dane fakty zostały zaobserwowane (lub wywnioskowane z innych faktów)

Istnieje właściwa dla reprezentacji wiedzy **procedura wnioskowania**, która sprawdza prawdziwość zadawanych pytań, w oparciu o aktualny stan bazy wiedzy i reguły wnioskowania.



Rys. 1.3: System z bazą wiedzy

1.2.2. Podsystem sterowania

Sterowanie systemu odpowiada za realizację zadanego celu (lub inaczej mówiąc, za rozwiązanie zadanego problemu). Uniwersalna postać sterowania to algorytm **przeszukiwania** przestrzeni stanów dla problemu. Istnieje szereg możliwych strategii przeszukiwania. Z grubsza możemy je podzielić na strategie **ślepego** przeszukiwania lub **poinformowanego** przeszukiwania (w sposób racjonalny - optymalny z punktu widzenia stosowanej funkcji użyteczności stanów).

System ekspertowy lub agentowy realizowany w technologii systemu z bazą wiedzy powinien posiadać następujące możliwości:

- reprezentować stany problemu i akcje agenta;
- integrować nowe obserwacje (lub nowo wprowadzone informacje);
- modyfikować wewnętrzną reprezentację świata, tzn. wnioskować o ukrytych własnościach świata wynikających z obserwacji;
- wybierać odpowiednie akcje agenta (rozwiązywać zadany problem).

Zwykle przyjmujemy, że sterowanie agenta komunikuje się z bazą wiedzy (KB) używając dwóch operacji TELL i ASK:

TELL: Agent → KB (powiedz bazie o nowych obserwacjach / faktach),

ASK : KB → Agent (zapytaj się bazy co robić).

Działanie agenta dysponującego bazą wiedzy przedstawia tabl. 1-2 – w postaci funkcji AgentZBaząWiedzy(). Podany funkcja odwołuje się do bazy wiedzy za pomocą funkcji TELL i ASK, przekazując im odpowiednie zdania wyrażone w języku reprezentacji wiedzy tej bazy. Generalnie są to zdania trzech rodzajów, tworzone przez poniższe podfunkcje:

- UtwórzZdanieObserwacji() – pobiera obserwację i indeks czasu a następnie generuje zdanie w języku bazy wiedzy reprezentujące obserwację w danej chwili czasu.
- UtwórzZapytanieAkcji() – pobiera indeks czasu i generuje zdanie w języku bazy wiedzy będące zapytaniem o to, jaką akcję należy wykonać.
- UtwórzZdanieAkcji() – generuje zdanie w języku bazy wiedzy reprezentujące wykonaną akcję w danej chwili czasu.

Konkretna realizacja tych funkcji zależy od typu reprezentacji wiedzy i własności realizowanego systemu sztucznej, takich jak wektor obserwacji, dostępne akcje, format pytań służących rozwiązaniu problemu.

Tabl. 1-2. Program agenta w systemie z bazą wiedzy

```
function AgentZBaząWiedzy(obserwacja) zwraca akcję
{ static:    KB, // baza wiedzy
             t;  // licznik (indeks czasu) – początkowo wynosi 1
     TELL(KB, UtwórzZdanieObserwacji(obserwacja,t));
     akcja ← ASK(KB, UtwórzZapytanieAkcji(t));
     TELL(KB, UtwórzZdanieAkcji(akcja, t));
     t ← t+1;
     return akcja;
}
```

Ważnym zagadnieniem systemu agentowego jest posiadanie zdolności do **uczenia się** (nabywania wiedzy o zasadach środowiska lub adaptowanie strategii działania).

Przedmiotem wykładu jest projektowanie **racjonalnie działających agentów,** ograniczone do ich komputerowej symulacji, popularnie określanych jako tzw. „softboty". Materiał wykładu podzielony został na trzy zasadnicze części: reprezentacja wiedzy i wnioskowanie, przeszukiwanie jako uniwersalny sposób rozwiązywania problemów, uczenie się wiedzy deterministycznej.

1.3. Język reprezentacji wiedzy

Omówimy teraz pokrótce podstawowe **języki reprezentacji wiedzy** w kilku ważnych aspektach, m.in. ontologii i epistemologii. **Ontologia** języka wskazuje na to czemu (jakim zjawiskom) odpowiadają symbole języka w rzeczywistym świecie. **Epistemologia** języka wyraża to, jak odbierane (oceniane) są te elementy świata - jakie wartości przyjmują jednostki wiedzy.

W tabeli 1-3 zebranych jest 5 istotnych języków. W niniejszym wykładzie szczegółowo omówimy projektowanie systemu z baza wiedzy wtedy, gdy baza wiedzy wyrażona jest w języku logiki lub pokrewnym.

Tabl. 1-3. Ontologia i epistemologia języka reprezentacji wiedzy

Język	Ontologia	Epistemologia
Rachunek zdań	Fakty	true/false/unknown
Logika predykatów (1 rzędu)	Fakty, obiekty, relacje	true/false/unknown
Logika temporalna	Fakty, obiekty, relacje (wiedza niepełna)	true/false/unknown
Teoria probabilistyczna	Fakty (wiedza niepewna)	Rozkład prawdopodobieństwa
Logika rozmyta	Fakty (wiedza niedokładna)	Stopień przynależności <0, 1>

Składnia języka L podaje reguły tworzenia poprawnych zdań języka (w logice predykatów nazywanych formułami). **Semantyka** języka L definiuje „znaczenie" zdań (formuł):

1. podaje ona znaczenie wszystkich symboli **X** języka L (czyli zawiera pewne odwzorowanie: **X**→ {elementy modelowanego świata}) i
2. sposób, w jaki zdaniom (formułom) można przypisać znaczenie, co z kolei pozwala określić ich wartość (np. logiczną).

Dla przykładu język działań arytmetycznych jest językiem logiki. Zasady składni mówią np., że (x+2 ≥ y) jest zdaniem, a (x2+y > {}) nie jest zdaniem tego języka. Z kolei zasady semantyki mówią np., że:

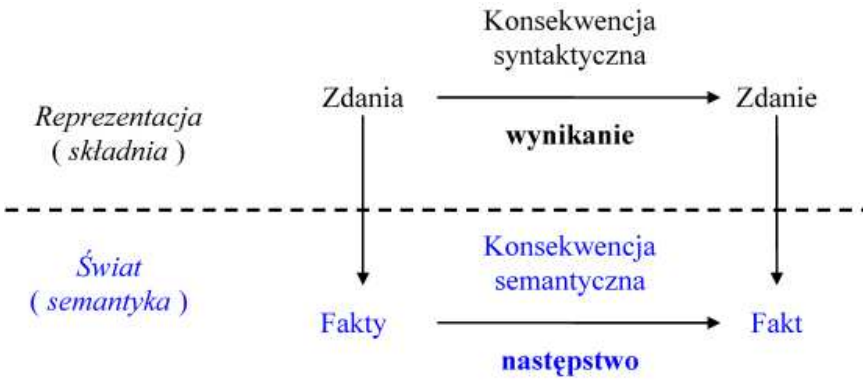
- x+2 ≥ y jest prawdziwe wtw. liczba x+2 jest nie mniejsza niż liczba y;
- x+2 ≥ y jest prawdziwe w świecie, w którym x = 7, y = 1;
- x+2 ≥ y jest fałszywe w świecie, w którym x = 0, y = 6.

1.4. System logicznego wnioskowania

1.4.1. Wynikanie zdań

Wynikanie (ang. *entailment*) jest związkiem pomiędzy zdaniami. Mówimy, że „ze zbioru zdań X wynika zdanie A" i oznaczamy to jako: X ╞ A , wtedy gdy odzwierciedla to następstwo (*konsekwencję semantyczną*) odpowiadających tym symbolom faktów w modelowanym świecie".

Zachodzi więc zależność pomiędzy elementami składni i semantyki języka, przedstawiona na rys. 1.4.



Rys. 1.4: Wynikanie zdań w języku logiki

Semantyczną relację **następstwa** wyrazimy poprzez relację zawierania się zbiorów reprezentujących realizacje świata, zwane **modelami**. Modele w logice to formalnie zdefiniowane **światy**, względem których można określać to co jest **prawdziwe** a co **nie**. **Model** dla zbioru zdań **X** to każdy **świat**, w którym prawdziwe są wszystkie zdania ze zboru **X**.

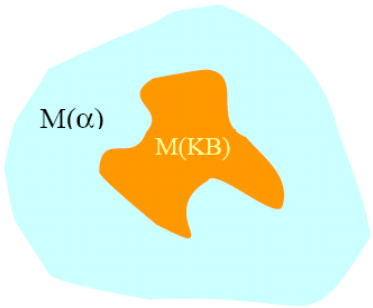
Możemy powiedzieć, że zdanie A **wynika** ze zbioru zdań **X**, co zaznaczamy $\mathbf{X} \models A$, jeśli A jest **prawdziwe** w każdym modelu dla **X**. Odnosząc to stwierdzenie do bazy wiedzy, która zawiera jedynie zdania uznane za prawdziwe w obserwowanym środowisku (realizacji świata) powiemy, że z bazy wiedzy KB **wynika** zdanie α wtw. gdy α jest **prawdziwe** dla wszystkich **modeli zdań** zawartych w KB (oznaczamy, $KB \models \alpha$).

Związek relacji wynikania (konsekwencji syntaktycznej) i następstwa (konsekwencji semantycznej) zapiszemy następująco (patrz rys. 1.5). Niech M(α) będzie zbiorem wszystkich modeli zdania α. Wtedy:

$$KB \models \alpha \text{ wtw. } M(KB) \subseteq M(\alpha).$$

Np. niech baza wiedzy KB zawiera zdania "Legia *wygrała mecz*" i "Wisła *wygrała mecz*". Z nich wynika zdanie: "Legia w*ygrała mecz lub Wisła wygrała mecz*", a to dlatego, że jest ono bardziej ogólne i jego model obejmuje modele obu poprzednich zdań.

Zdania uznane za **równoważne** w pewnej dziedzinie wiedzy są specyficznym przypadkiem wynikania. Np. ze zdania (x+y= 4) wynika, że (4 = x+y), a w *matematyce* są to zdania równoważne



Rys. 1.5: Konsekwencja semantyczna jako relacja zawierania się modeli zdań.

1.4.2. Wnioskowanie

Celem **procesu wnioskowania** jest ujawnienie lub sprawdzenie tego, czy zachodzi zależność pomiędzy zdaniami języka, zwana wynikaniem. Ogólnie rzecz biorąc, każdy proces **wnioskowania** (ang. *inference*) w systemie logicznym występuje w jednej z dwóch postaci:

1) Proces **wyprowadzania** (generowania) nowych zdań ze zdań przyjętych za prawdziwe (tzn. reprezentujących prawdziwe fakty).

2) Proces **sprawdzenia** (dowód), czy zadanie zdanie *A* (pytanie) wynika ze zbioru zdań ***X***, tzn. czy zachodzi $\boldsymbol{X} \models A$.

Dla danego języka logicznego możemy zdefiniować pojęcie **wyprowadzalności zdań** (konsekwencja syntaktyczna, *derivability*):

„zdanie *A* jest wyprowadzalne ze zbioru zdań ***X*** przy użyciu procedury wnioskowania (dowodzenia) Π, co oznaczymy jako $\boldsymbol{X} \vdash_{\Pi} A$, wtw. gdy Π wyprowadza (znajduje dowód zdania) *A* ze zdań zbioru ***X***."

Podobnie oznaczenie $KB \vdash_i \alpha$ powie nam, że zdanie α jest wyprowadzalne z bazy wiedzy KB przy użyciu procedury *i.*

Interesują nas tylko takie procedury wnioskowania, które są **poprawne** i **zupełne** :

- Poprawność procedury wnioskowania

  Procedura wnioskowania Π jest poprawna wtedy i tylko wtedy (wtw.) gdy dla każdego zbioru zdań ***X*** i każdego zdania *A*: $\boldsymbol{X} \vdash_{\Pi} A$ pociąga za sobą $X \models A$.

  Innymi słowy, jeśli poprawna procedura wnioskowania wskazuje na istnienie relacji wynikania pomiędzy pewnymi zdaniami, to ona zawsze rzeczywiście zachodzi.

- Zupełność procedury wnioskowania

  Procedura wnioskowania Π jest zupełna wtw. gdy dla każdego zbioru zdań ***X*** i każdego zdania *A*: $\boldsymbol{X} \models A$ pociąga za sobą $\boldsymbol{X} \vdash_{\Pi} A$.

  Tym samym wskazujemy, że zupełna procedura wnioskowania to taka, która jest w stanie wykazać każde rzeczywiście istniejące wynikanie.

1.4.3. System logiczny

Zaletą **systemu logicznego wnioskowania** (systemu logiki lub po prostu **logiki**) jest to, że udostępnia on język formalny do takiej reprezentacji informacji, z której można wyciągać wnioski. Dlatego też obok składni i semantyki języka logiki, czyli typowych elementów opisu każdego języka, w naturalny sposób dołączamy do niego mechanizmy wynikania i wnioskowania.

Podsumowując, logika obejmuje język reprezentacji wiedzy (charakteryzowany poprzez składnię i semantykę) oraz odpowiedni mechanizm wnioskowania (dedukcji). W następnych rozdziałach zdefiniujemy dwa podstawowe systemy logiczne (*rachunek zdań*, *logika pierwszego rzędu*), które są wystarczająco „mocne", aby wyrazić większość interesujących nas rzeczy i dla których istnieją poprawne i zupełne procedury wnioskowania.

Dla każdego języka logiki, można zaproponować wiele procedur wnioskowania – sam język nie wyznacza unikalnego mechanizmu wnioskowania, którym można się posługiwać. Jednak typowy system logicznego wnioskowania polega na stosowaniu **reguł wnioskowania**, czyli formuł zawsze prawdziwych (tautologie języka).

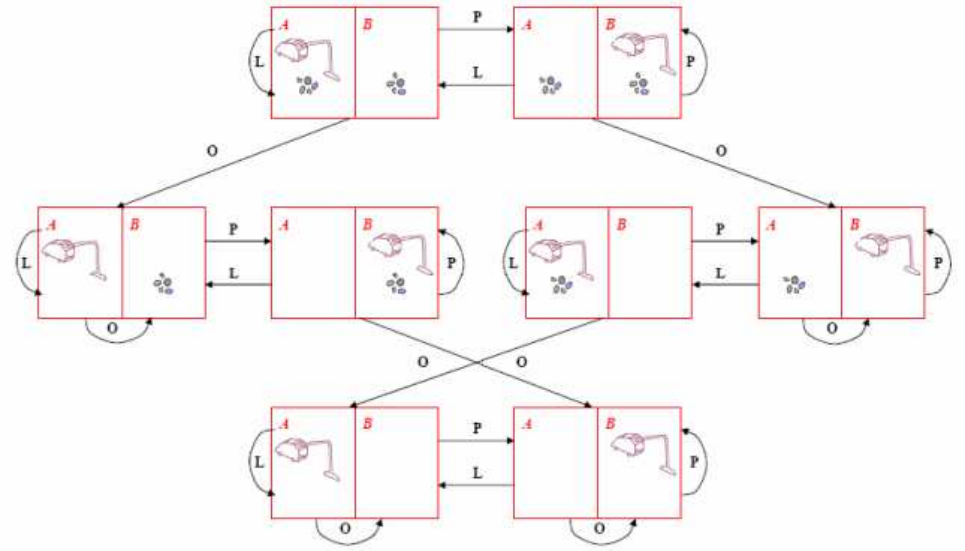
1.5. Przeszukiwanie

Przeszukiwanie przestrzeni stanów problemu oznacza, że system agentowy wybiera akcje (podejmuje decyzje) w oparciu o aktualny stan problemu i zadany cel. Sposób wyboru zależy od konkretnej strategii przeszukiwania, realizowanej przez agenta.

Przykład. Przestrzeń stanów dla problemu „agenta odkurzającego" (rys. 1.6)

Stany wyznaczone są łącznie przez stopnie zabrudzenia obu kratek i pozycję odkurzacza. Wystarczą nam trzy akcje: [*w lewo* (*L*), *w prawo* (*P*), odkurzanie *(O)*]. Poprzednio wymienioną akcję „NicNieRób" reprezentujemy poprzez akcję „w lewo" będąc w kratce A lub akcję „w prawo" będąc w kratce B. W obu przypadkach efekt jest taki sam, jak efekt akcji „NicNieRób". Zmniejszenie liczby

akcji do 3 nie wpływa na liczbę stanów problemu. Warunek stopu to: „obie kratki są czyste (brak brudu)”. Niech koszt akcji wynosi zawsze 1 za każdą akcję.



Rys. 1.6: „Agent odkurzający” i jego przestrzeń stanów

Na potrzeby przeszukiwania reprezentacja stanów jest zwykle prostą strukturą danych, zawierającą dane potrzebne jedynie dla *generacji następnika, funkcji oceny* i *warunku stopu.* Reprezentacja akcji ma postać procedury, która generuje następne stany problemu. Reprezentacja celu jest to zwykle pośrednia informacja o celu, wyrażona poprzez warunek stopu i funkcję oceny kosztów resztkowych (heurystykę).

Poza powyższą informacją, z punktu widzenia strategii przeszukiwania, stan nie niesie żadnej dalszej informacji - ma zwykle charakter „czarnej skrzynki”. Taka reprezentacje problemu nie daje żadnych dalszych wskazówek co do wyboru akcji, np. zależnych od dokładniejszej charakterystyki stanu problemu. **Planowanie** ma na celu „usprawnienie” przeszukiwania przez „otwarcie” reprezentacji stanów, celów i akcji. Plan jest zwykle sekwencją akcji wiodącą agenta do celu bez potrzeby kolejnych obserwacji lub do podcelu, w którym wymagana jest dalsza percepcja stanu środowiska.

1.6. Uczenie

W potocznym znaczeniu pojęcie „uczenie się” jest często utożsamiane z eksploracją nieznanego środowiska i z nabywaniem wiedzy będącej **generalizacją** tych obserwacji – określającej zasady rządzące w środowisku. Jest to nieodzowny element działania agenta wtedy, gdy projektant systemu nie dysponuje pełną informacją o środowisku.

Przez „uczenie się” rozumiemy też uczenie się **sposobu wyboru akcji**, czyli uczenie funkcji decyzyjnej. Osiągamy to wystawiając system na oddziaływanie realnego środowiska, zamiast od razu w pełni specyfikować jego zasady działania. Zadaniem procesu uczenia jest zmiana mechanizmu decyzyjnego systemu w celu poprawy skuteczności jego działania. Aby rozwiązać sam problem uczenia możemy zastosować przeszukiwanie przestrzeni problemu uczenia z wykorzystaniem specyficznych funkcje użyteczności dla oszacowania skuteczności działania systemu po zmianie.

Projekt modułu uczącego zależy od tego jak reprezentujemy funkcję systemu i jaki jej element jest modyfikowany, a także jaki rodzaj sprzężenia zwrotnego stosujemy podczas uczenia. O wybranych funkcjach decyzyjnych i ich procedurach uczenia powiemy później. Natomiast sprzężenie zwrotne przyjmuje zwykle jedną z trzech poniższych postaci:

1. Uczenie z nadzorem (ang. *supervised learning*): istnieją prawidłowe (wzorcowe) odpowiedzi dla każdego przykładu uczącego;
2. Uczenie ze wzmacnianiem (ang. *reinforcement learning*): brak wzorcowej odpowiedzi ale istnieje krytyk nagradzający ”dobre” akcje (zbliżające agenta do prawidłowego stanu);
3. Uczenie bez nadzoru (ang. *unsupervised learning*): brak wzorcowych odpowiedzi, brak krytyka.

1.7. Pytania

1. Wyjaśnić cztery definicje pojęcia „inteligencji”.
2. Jakie są zasadnicze **zastosowania** Sztucznej Inteligencji w informatyce?
3. Omówić zasadę pracy **systemu z bazą wiedzy** i główne funkcje komunikowania się sterowania z bazą wiedzy.
4. Wyjaśnić pojęcia: „**wynikanie”** i „**wnioskowanie”** zdań. Zilustrować odpowiedź na przykładach ze świata matematyki
5. Omówić pojęcie **przeszukiwania przestrzeni stanów** jako uniwersalnego sposobu rozwiązywania problemów.
6. Jakie są główne **paradygmaty uczenia** w systemie Sztucznej Inteligencji ?

1.8. Zadania

Zad. 1.1

Dla „agenta odkurzającego” (z pamięcią) wyznaczyć sekwencję akcji wiodącą do celu „wszędzie czysto” w trzech różnych sytuacjach:

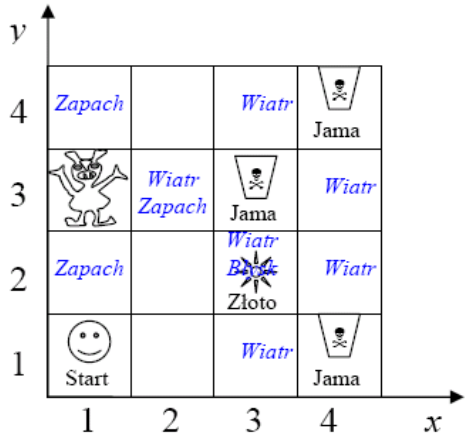
1) Środowisko jest deterministyczne i w pełni obserwowalne → problem jednostanowy (1 stan problemu to 1 stan środowiska).

2) Środowisko jest deterministyczne ale nie w pełni obserwowalne - na skutek braku pewnych czujników → problem wielostanowy – niepewność aktualnego stanu środowiska (1 stan problemu to wiele możliwych stanów środowiska).

3) Środowisko jest niedeterministyczne (problem ewentualności) – niepewność następnego stanu środowiska. Obserwacje dostarczają nowych informacji o środowisku.

Zad. 1.2

„*Świat Wumpusa*” to wczesna gra komputerowa (rys. 1.7). Celem agenta jest znalezienie złota i wydostanie się z jaskini (powrót do kwadratu startowego [1,1]). Miara skuteczności agenta: złoto: +1000 pkt., śmierć: -1.000 pkt, -1 pkt. za każdą akcję, -10 pkt. za skorzystanie z jedynej strzały. Nie zawsze agent może uzyskać dodatni wynik - świat może być „źle” zdefiniowany: złoto może być w jamie lub być niedostępne – otoczone *jamami* i/lub *Wumpusem*.

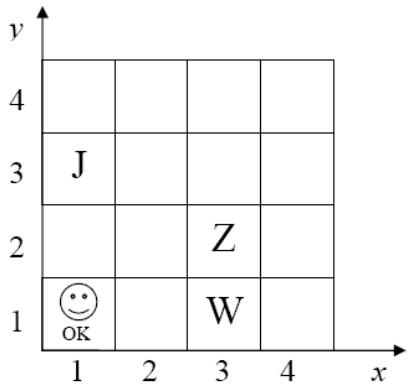


Rys. 1.7: Struktura „świata Wumpusa”.

Obserwacje tworzą wektor 5-elementowy: [zapach, **wiatr**, **błysk**, **uderzenie**, **krzyk**]. W kwadracie, gdzie mieszka Wumpus oraz w ściśle przylegających agent czuje zapach (*smród Wumpusa*); W kwadratach przylegających do jamy agent czuje wiatr. W kwadracie, gdzie znajduje się złoto agent obserwuje błysk; Jeśli agent wejdzie na ścianę to czuje uderzenie; Kiedy Wumpus zostaje zabity, w całej jaskini słychać krzyk.

Akcje: { *ObrótWLewo,* ObrótWPrawo*, RuchWPrzód, Podnieść, Upuścić, Strzelić, Wyjść, Zginąć* }. *„Strzelić”* - zużywa jedyną strzałę w kierunku patrzenia agenta; *„Podnieść”* złoto, jeśli jest w tej samym kwadracie; *„Upuścić”* złoto, pozostawiając je w tym samym kwadracie; „Wyjść” - pozwala agentowi opuścić jaskinię, o ile znajduje się on w kwadracie startowym. „Zginąć” - agent ginie jeśli wejdzie na kwadrat, w którym znajduje się *Wumpus* lub *jama*.

Przedstawić sekwencję obserwacji i akcji agenta oraz (stanu rozpoznania środowiska) wiodącą go do kratki ze złotem w poniższej wersji świata (rys. 1.8) (na początku zupełnie nieznanego agentowi). Początkowa obserwacja w kwadracie [*x*,*y*]=[1,1]: agent nie odczuwa zapachu ani wiatru.



Rys. 1.8: Stan początkowy „świata Wumpusa”.

Oznaczenia: ☺ oznacza położenie agenta. OK – kratka jest „bezpieczna”. **J** – jama, **W** – Wumpus, **Z** – złoto.

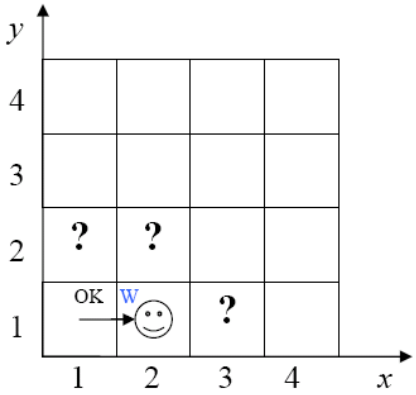
Zad. 1.3

W *świecie Wumpusa* załóżmy sytuację powstałą po:(1) pustej obserwacji w [1,1], ruchu w prawo do [2,1] i (2) obserwacji wiatru w [2,1].

Teraz agent chciałby wiedzieć, czy w sąsiednich kratkach: [1,2], [2,2] i [3,1], występują jamy. Sprawdzić wynikanie, lub nie, poniższych zdań z aktualnej bazy wiedzy, poprzez sprawdzenie zawierania się modeli, dla zdań:

A) α1= „w kratce [1,2] nie ma jamy”

B) α2= „w kratce [2,2] nie ma jamy”



Rys. 1.8: Stan „świata Wumpusa” po 2 obserwacjach.

2. Rachunek zdań

Prezentację klasycznych języków logiki rozpoczniemy od najprostszego z nich, zwanego **rachunkiem zdań**.

2.1. Składnia rachunku zdań

W skład alfabetu rachunku zdań wchodzą:

- stałe logiczne *True* i *False*,
- symbole zdaniowe (*P*, *Q*, *R*,...),
- spójniki logiczne (koniunkcja, alternatywa, implikacja, równoważność, negacja) $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$, $\neg$
- nawiasy okrągłe **(**, **)**.

Z liter alfabetu tworzone są **zdania** za pomocą następujących zasad:

- *True* , *False* i symbole zdaniowe są zdaniami atomowymi.
- Jeśli *A* i *B* są zdaniami to $(A \wedge B)$, $(A \vee B)$, $(A \Rightarrow B)$ i $(A \Leftrightarrow B)$ też są zdaniami języka.
- Jeśli *A* jest zdaniem to $\neg A$ też jest zdaniem.

Mianem **literału** określamy zdanie atomowe lub negację zdania atomowego.

2.2. Semantyka rachunku zdań

Semantyka rachunku zdań może być formalnie ujęta jako struktura algebraiczna: <D, φ >. **Dziedzina** D to zbiór faktów, do których odnoszą się symbole języka. **Interpretacja** φ to funkcja przyporządkowująca symbolom zdaniowym fakty należące do dziedziny i wartościująca zdania jako *Prawdę* (*True*) lub *Fałsz* (*False*). Wartościowanie zdań atomowych zależy od prawdziwości reprezentowanego faktu w aktualnym świecie a przy wartościowaniu zdań złożonych korzystamy z tabelki prawdy dla spójników (0 oznacza *Fałsz* a 1 – *Prawdę*) (rys. 2.1).

Model zdania A (oznaczmy go jako M(A) = <D, φ >) jest wyznaczony przez każdą interpretację φ dla dziedziny D, w której zdanie A jest prawdziwe. **Tautologia** w dziedzinie D jest to zdanie prawdziwe w każdej interpretacji (modelu) dla D.

Rachunek zdań jest **rozstrzygalny**, tzn. istnieje algorytm, który dla dowolnego zdania A stwierdza, czy A jest tautologią. Takim algorytmem jest chociażby sprawdzanie tabelki prawdy dla zdania. Ponieważ każde zdanie zawiera skończoną liczbę symboli i spójników, więc tabelka będzie miała skończony rozmiar co umożliwi sprawdzenie wszystkich wierszy.

P	Q	$\neg$P	P$\wedge$Q	P$\vee$Q	P$\Rightarrow$Q	P$\Leftrightarrow$Q
0	0	1	0	0	1	1
0	1	1	0	1	1	0
1	0	0	0	1	0	0
1	1	0	1	1	1	1

Rys. 2.1: Tabelka prawdy dla spójników logicznych

Dla przykładu, dzięki tabeli prawdy pokazujemy, że zdanie złożone, $((P \vee H) \wedge \neg H) \Rightarrow P$, jest tautologią, tzn. zawsze prawdziwą niezależnie od modelu dla P i H (rys. 2.2) - zauważmy, że ostatnia kolumna jest zawsze prawdziwa.

P	H	P∨H	(P∨H)∧¬H	((P∨H)∧¬H)⇒P
0	0	0	0	1
0	1	1	0	1
1	0	1	1	1
1	1	1	0	1

Rys. 2.2: Sprawdzanie tautologii przy pomocy tabelki prawdy.

2.3. Wnioskowanie

2.3.1. Wnioskowanie z tabelą prawdy

Procedura sprawdzająca wszystkie modele jest procedurą wnioskowania **poprawną** i **zupełną**. Może ona zostać efektywnie zaimplementowana wtedy, gdy formuły dają się uporządkować (np. odpowiednio do odległości miejsca od pozycji startowej agenta, do którego te formuły się odnoszą).

Np. funkcja WNIOSKUJTABPRAWDY() (tab. 2.1) jest efektywną implementacją procedury wnioskowania. Korzysta ona z rekursywnej funkcji TABPRAWDY(), wywoływanej dla częściowego i stopniowo rozszerzanego modelu. Podfunkcja CZYMODEL() sprawdza poprawność zdania (lub bazy wiedzy) w częściowym modelu a podfunkcja ROZSZERZ() rozszerza model o wartość dla kolejnego symbolu (zdania).

Tabl. 2-1. Procedura wnioskowania polegająca na sprawdzaniu tabelki prawdy

```
funkcja WNIOSKUJZTABPRAWDY(KB, α)
zwraca wynik: true lub false
{  symbole ← symbole zdaniowe w KB i α ;
   return TABPRAWDY(KB, α , symbole, []);
}
```

```
funkcja TABPRAWDY(KB, α, symbole, model)
zwraca wynik: true lub false
{ if (symbole == ∅ )  {
        if CZYMODEL(KB, model) return CZYMODEL(α, model);
        else return true;
  } else {
    P ← Pierwszy(symbole); reszta ← Reszta(symbole);
    return (TABPRAWDY(KB, α, reszta, Rozszerz(P, true, model)
       && TABPRAWDY(KB, α, reszta, Rozszerz(P, false,model)); }
}
```

2.3.2. Twierdzenia o dedukcji

Wiemy już, że zdanie logiczne jest tautologią wtw. gdy jest prawdziwe we wszystkich modelach języka. Np. tautologiami są następujące zdania:

$True, \qquad A \vee \neg A, \qquad A \Rightarrow A, \qquad (A \wedge (A \Rightarrow B)) \Rightarrow B$

Tautologia jest powiązana z wnioskowaniem poprzez pierwsze **twierdzenie o dedukcji**, które mówi, że:

$KB \models \alpha$ wtw. gdy formuła $(KB \Rightarrow \alpha)$ jest tautologią.

To twierdzenie prowadzi do metod wnioskujących wprost.

Wiemy też, że formuła A jest spełnialna wtw. gdy posiada przynajmniej jeden model. Wtedy niespełnialną formułą jest taka, która nie posiada żadnego modelu. Np.:

$A \wedge \neg A$ jest niespełnialne.

Spełnialność formuły jest powiązana z wnioskowaniem poprzez drugie **twierdzenie o dedukcji**:

$KB \models \alpha$ wtw. gdy formuła $(KB \wedge \neg \alpha)$ jest niespełnialna.

To twierdzenie prowadzi do wnioskowania nie wprost – do dowodu przez zaprzeczenie.

2.3.3. Przekształcenia równoważne

Na potrzeby procedur wnioskowania zdania w bazie wiedzy powinny przyjmować wymagane postacie *normalne*. Dla uzyskania takich postaci można stosować przekształcenia zdania do równoważnych zdań.

Istnieje szereg równoważnościowych przekształceń zdań logicznych. Mówimy, że dwa zdania są **logicznie równoważne** (oznaczamy ≡) wtw. gdy są poprawne w tych samych modelach. Innymi słowy logiczną równoważność możemy powiązać z wynikaniem:

$$\alpha \equiv ß \quad \text{wtw.} \quad (\alpha \models \beta) \text{ i } (\beta \models \alpha)$$

Oto podstawowe pary logicznie równoważnych zdań:

1. $(\alpha \wedge ß) \equiv (ß \wedge \alpha)$ – przemienność $\wedge$
2. $(\alpha \vee ß) \equiv (ß \vee \alpha)$ – przemienność $\vee$
3. $((\alpha \wedge ß) \wedge \lambda) \equiv (\alpha \wedge (ß \wedge \lambda))$– łączność $\wedge$
4. $((\alpha \vee ß) \vee \lambda) \equiv (\alpha \vee (ß \vee \lambda))$– łączność $\wedge$
5. $\neg(\neg \alpha) \equiv \alpha$ – eliminacja podwójnej negacji
6. $(\alpha \Rightarrow ß) \equiv (\neg ß \Rightarrow \neg \alpha)$ – kontrapozycja
7. $(\alpha \Rightarrow ß) \equiv (\neg \alpha \vee ß)$ – eliminacja implikacji
8. $(\alpha \Leftrightarrow ß) \equiv ((\alpha \Rightarrow ß) \wedge (ß \Rightarrow \alpha)$ – eliminacja równoważności
9. $\neg(\alpha \wedge ß) \equiv (\neg \alpha \vee \neg ß)$ – prawo de Morgana
10. $\neg(\alpha \vee ß) \equiv (\neg \alpha \wedge \neg ß)$ – prawo de Morgana
11. $(\alpha \wedge (ß \vee \lambda)) \equiv ((\alpha \wedge ß) \vee (\alpha \wedge \lambda))$ – rozdzielczość $\wedge$ względem $\vee$
12. $(\alpha \vee (ß \wedge \lambda)) \equiv ((\alpha \vee ß) \wedge (\alpha \vee \lambda))$ – rozdzielczość $\vee$ względem $\wedge$

2.3.4. Reguły wnioskowania

Dla rachunku zdań istnieją zupełne i poprawne **procedury wnioskowania** (dowodzenia formuł). Stosują one **reguły wnioskowania** o ogólnej postaci:

$$\frac{poprzednik}{nastepnik}$$

Reguła wnioskowania mówi, że jeśli spełniony jest warunek zadany *poprzednikiem* reguły to wnioskowana jest poprawność *następnika*. Oczywiście, aby móc stosować jakąś regułę wnioskowania musimy upewnić się, że jest ona poprawna we wszystkich modelach języka, tzn. że jest tautologią języka. Powiemy, że reguła wnioskowania o postaci

$$\frac{\alpha_1, \alpha_2, \cdots, \alpha_n}{\beta}$$

jest **poprawna**, jeśli zdanie β jest prawdziwe w każdej interpretacji (modelu), w której prawdziwe są zdania: $\alpha_1, \alpha_2, \ldots, \alpha_n$.

Wybór poprawnych i najczęściej stosowanych reguł wnioskowania:

1. Reguła **odrywania** (*modus ponens*):

$$\frac{\alpha, \quad \alpha \Rightarrow \beta}{\beta}$$

2. Reguła **eliminacji koniunkcji**:

$$\frac{\alpha_1 \wedge \cdots \wedge \alpha_n}{\alpha_i}$$

3. Reguła **wprowadzania koniunkcji**:

$$\frac{\alpha_1, \cdots, \alpha_n}{\alpha_1 \wedge \cdots \wedge \alpha_n}$$

4. Reguła **rezolucji**:

$$\frac{\alpha \vee \beta, \quad \neg\beta \vee \gamma}{\alpha \vee \gamma}$$

Sprawdźmy za pomocą tablicy prawdy poprawność powyższej reguły rezolucji (rys. 2.3).

α	β	γ	$\alpha \vee \beta$	$\neg\beta \vee \gamma$	$\alpha \vee \gamma$
0	0	0	0	1	0
0	0	1	0	1	1
0	1	0	1	0	0
0	1	1	1	1	1
1	0	0	1	1	1
1	0	1	1	1	1
1	1	0	1	0	1
1	1	1	1	1	1

Rys. 2.3: Tabela prawdy dla reguły rezolucji

W powyższej tabeli wyróżniono na różowo 4 wiersze, dla których spełniony jest poprzednik tej reguły. Widzimy, że w tych wierszach następnik również jest zawsze spełniony. Tym samym reguła rezolucji jest poprawna.

2.4. Postacie normalne

Procedury wnioskowania zakładają, że formuły występują we właściwej postaci **normalnej**. Pozwala to zdefiniować obliczeniowo efektywną procedurę wnioskowania.

2.4.1. Klauzula Horna i reguła odrywania

Dla procedur stosujących regułę *Modus Ponens* (zwaną także *regułą odrywania*) zdania (lub formuły) przyjmują postać tzw. **klauzul Horna**. Klauzula Horna to pojedynczy prosty literał lub implikacja o postaci:

(*koniunkcja prostych literałów*) $\Rightarrow$ prosty literał .

„Prosty" oznacza „pozytywny", nie zanegowany. Np. w poniższym zdaniu występują trzy klauzule Horna: $C \wedge (B \Rightarrow A) \wedge (C \wedge D \Rightarrow B)$

Reguła odrywania (*Modus Ponens*) stosowana jest w procedurach wnioskowania dla rachunku zdań wtedy, gdy zdania w bazie wiedzy występują w postaci klauzul Horna. Tworzą one poprzednik reguły:

$$\frac{\alpha_1, \cdots, \alpha_n, \qquad \alpha_1 \wedge \cdots \wedge \alpha_n \Rightarrow \beta}{\beta}$$

Z reguły odrywania korzystają procedury wnioskowania wprost: **progresywna** (wnioskowanie w przód) i **regresywna** (wnioskowanie wstecz).

2.4.2. Postać normalna CNF i reguła rezolucji

Drugą postacią normalną dla zdań (lub formuł) jest **koniunkcyjna postać normalna** (ang. *Conjunctive Normal Form*, **CNF**) będąca koniunkcją alternatyw literałów.

Np.: w rachunku zdań poniższe zdanie jest w postaci CNF:

$$(A \vee \neg B) \wedge (B \vee \neg C \vee \neg D)$$

Postać CNF jest odpowiednia dla procedur wnioskowania korzystających z reguły rezolucji, czyli dla wnioskowania nie wprost, inaczej mówiąc: dowodzenia zdania zapytania przez zaprzeczenie.

Reguła rezolucji w rachunku zdań jest postaci:

$$\frac{l_1 \vee \ldots \vee l_k \,,\, m_1 \vee \ldots \vee m_n}{l_1 \vee \ldots \vee l_{i-1} \vee l_{i+1} \vee \ldots \vee l_k \vee m_1 \vee \ldots \vee m_{j-1} \vee m_{j+1} \vee \ldots \vee m_n}$$

gdzie l_i i m_j są komplementarnymi literałami ($l_i = \neg m_j$).

Np.: jeśli zachodzą zdania: $P_{13} \vee P_{22}$ i $\neg P_{22}$ to wnioskujemy stąd, że zachodzi P_{13} .

Pokażemy poprawność reguły rezolucji stosując jedynie przekształcenia zdań.

1) Zakładamy, że zachodzi poprzednik. Stosując równoważność, $\alpha \Rightarrow \beta \equiv \neg\alpha \vee \beta$, przejdziemy z postaci po prawej stronie do lewej strony dla obu zdań w poprzedniku. Wyłączamy przy tym komplementarne literały od reszty literałów. Otrzymujemy:

$\neg(l_i \vee \ldots \vee l_{i-1} \vee l_{i+1} \vee \ldots \vee l_k) \Rightarrow l_i$

$$\neg m_j \Rightarrow (m_1 \vee \ldots \vee m_{j-1} \vee m_{j+1} \vee \ldots \vee m_n)$$

2) Z założenia, $l_i = \neg m_j$, i z przechodniości implikacji, $\alpha \Rightarrow \beta \Rightarrow \gamma$, wynika:

$\neg(l_i \vee \ldots \vee l_{i-1} \vee l_{i+1} \vee \ldots \vee l_k) \Rightarrow (m_1 \vee \ldots \vee m_{j-1} \vee m_{j+1} \vee \ldots \vee m_n)$

3) Ponownie stosujemy równoważność z pkt. 1) ale teraz przekształcamy implikację z pkt. 2) na postać alternatyw literałów:

$$l_i \vee \ldots \vee l_{i-1} \vee l_{i+1} \vee \ldots \vee l_k \vee m_1 \vee \ldots \vee m_{j-1} \vee m_{j+1} \vee \ldots \vee m_n .$$

Tym samym uzyskaliśmy postać następnika reguły rezolucji co kończy dowód.

Przykład.

Konwersja zdania do postaci normalnej CNF. Przekształcimy zdanie, $B_{11} \Leftrightarrow (P_{12} \vee P_{21})$.

1. Usuwamy równoważność $\Leftrightarrow$, zamieniając $\alpha \Leftrightarrow \beta$ na $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$.

$$(B_{1,1} \Rightarrow (P_{1,2} \vee P_{2,1})) \wedge ((P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1})$$

2. Usuwamy obie implikacje $\Rightarrow$, zamieniając $\alpha \Rightarrow \beta$ na $\neg\alpha \vee \beta$.

$$(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg(P_{1,2} \vee P_{2,1}) \vee B_{1,1})$$

3. Wprowadzamy $\neg$ do środka nawiasów stosując reguły de Morgana i ewentualnie eliminujemy podwójną negację:

$$(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge ((\neg P_{1,2} \wedge \neg P_{2,1}) \vee B_{1,1})$$

4. Stosujemy prawo rozdzielczości ($\wedge$ nad $\vee$) i rozpisujemy:

$$(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg P_{1,2} \vee B_{1,1}) \wedge (\neg P_{2,1} \vee B_{1,1})$$

Uzyskaliśmy zdanie w postaci CNF.

2.5. Wnioskowanie poprzez rezolucję (dowód nie wprost)

Procedury wnioskowania (dowodzenia) zdań stosujące regułę rezolucji prowadzą **dowód przez zaprzeczenie**, tzn. aby dowieść, że zachodzi

wynikanie ($KB \models \alpha$)

pokazują one, że

zdanie ($KB \wedge \neg\alpha$) jest niespełnialne.

Schemat algorytmu wnioskowania przez rezolucję pokazuje tabela 2-2.

Tab. 2-2. Procedura wnioskowania przez rezolucję w rachunku zdań

```
funkcja Rezolucja(KB, α) zwraca wynik: True lub False
{
  zdania ← zbiór zdań w postaci CNF dla (KB ∧ ¬α);
  nowe ← {};
  while ( True ) {
     for each (Ci, Cj ∈ zdania ) {
        rezolwenty ← KrokRezolucji(Ci, Cj);
        if (rezolwenty zawierają zdanie puste) return True;
        nowe ← nowe ∪ rezolwenty;
     }
     if (nowe ⊆ zdania ) return False;
     zdania ← zdania ∪ nowe;
  }
}
```

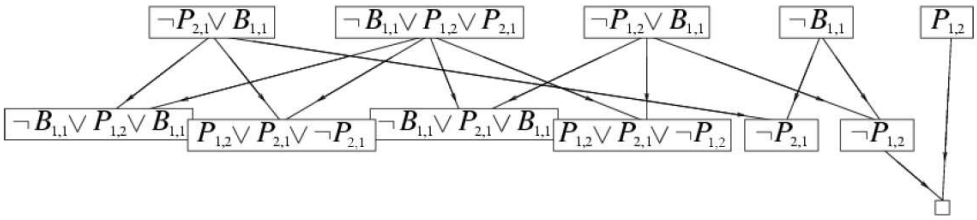
W funkcji Rezolucja() najpierw zamieniamy zdania w bazie wiedzy rozszerzonej o negację zapytania, (KB $\wedge\neg\alpha$), na postać CNF. Następnie w pętli, dla każdej odpowiedniej pary zdań, funkcja KrokRezolucji() generuje *rezolwentę* swoich 2 argumentów, tzn. zdanie w następniku reguły rezolucji, które jest pozbawione pary komplementarnych symboli. Są możliwe dwa warunki zakończenia procedury:

1) nie można dodać nowych zdań, wtedy α jest fałszywe w modelu M(KB), lub

2) w wyniku rezolucji powstaje zdanie puste, co wynika ze sprzeczności w bazie; a to oznacza, że α jest prawdziwe w modelu M(KB).

Przykład wnioskowania metodą rezolucji.

Niech baza wiedzy agenta zawiera m.in..: $B_{11} \Leftrightarrow (P_{1,2} \vee P_{2,1})$, i $\neg B_{11}$. Chcemy dowieść, że zachodzi zdanie: $\alpha = \neg P_{12}$. Na rys. 2.4. górny rząd zawiera zdania powstałe po normalizacji $(KB \wedge \neg \alpha)$ do postaci CNF. Dolny rząd zawiera *rezolwenty* par zdań zawierających komplementarne literały. Kolejny rząd zawiera zdanie puste powstałe z rezolucji pary P_{12} i $\neg P_{12}$. Dowodzi to, że zdanie zapytania jest spełnione w modelu bazy wiedzy, tzn. $KB \models \alpha$.



Rys. 2.4: Przykład kroków rezolucji [6]. Wygenerowana zostanie formuła pusta.

2.6. Wnioskowanie z regułą odrywania (generacja lub dowód wprost)

Procedury wnioskowania korzystające z reguły odrywania stosują dowodzenie wprost. Wyróżnimy tu zasadniczo dwa sposoby takiego wnioskowania:

- procedura progresywna (wnioskowanie w przód) – generowanie zdań – i
- procedura regresywna (wnioskowanie wstecz) – dowód wprost.

2.6.1. Wnioskowanie w przód

Idea procedury wnioskowania **progresywnego (w przód)**:

1. wykonaj każdą regułę, której warunek (poprzednik) jest spełniony w KB,
2. dodaj wynik wyprowadzenia (następnik reguły) do KB,
3. kontynuuj kroki 1-2 aż do znalezienia zdania zapytania lub niemożliwości wygenerowania nowych zdań.

Progresywna procedura wnioskowania jest poprawna i zupełna dla bazy wiedzy o postaci klauzul Horna.

Przykład wnioskowania w przód.

Dane są zdania w KB w postaci klauzul Horna:

$p \Rightarrow q$

$m \wedge n \Rightarrow p$

$Ewa \wedge m \Rightarrow n$

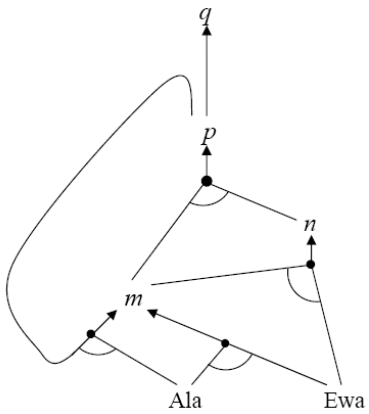
$Ala \wedge p \Rightarrow m$

$Ala \wedge Ewa \Rightarrow m$

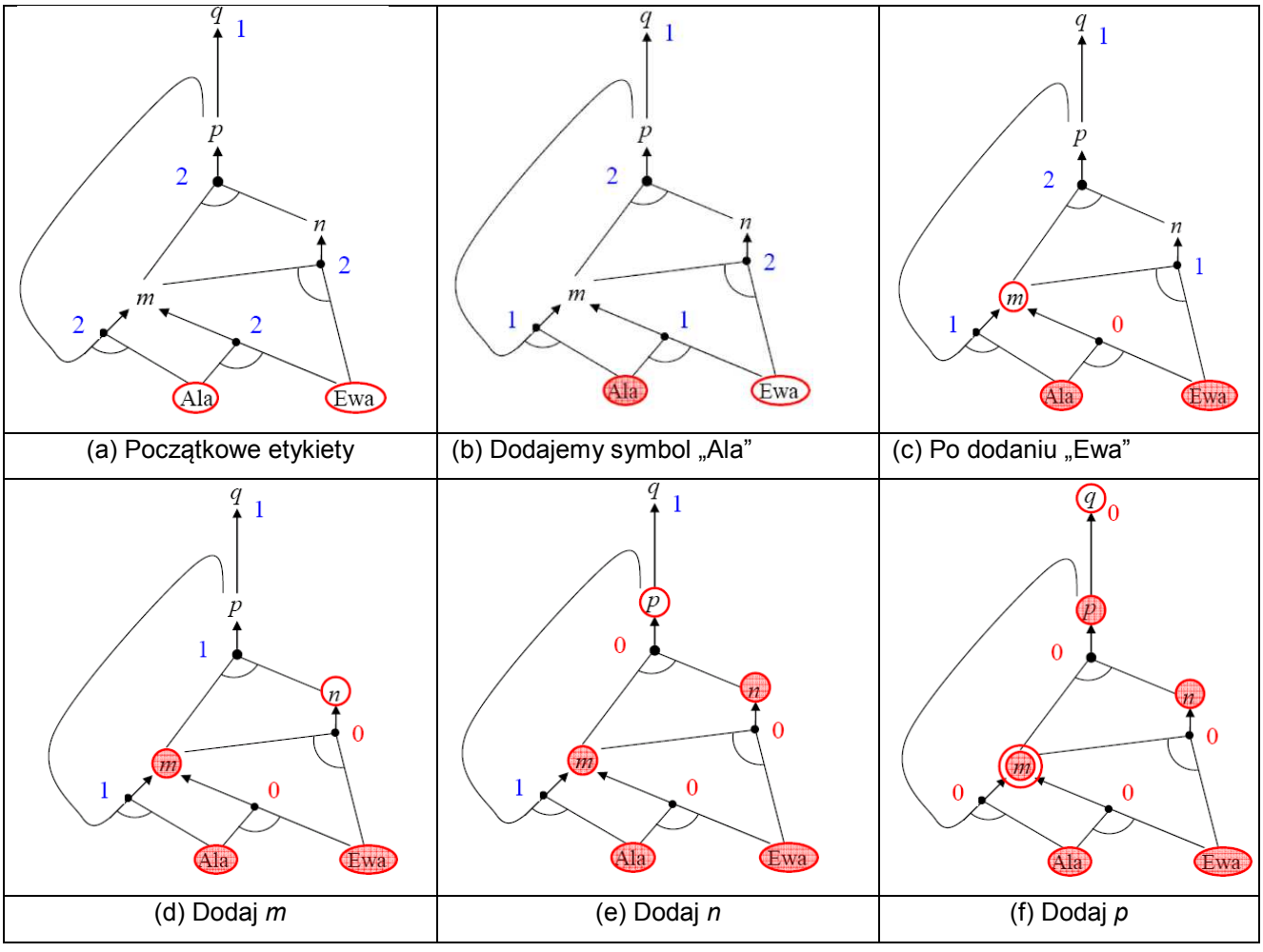
Ala

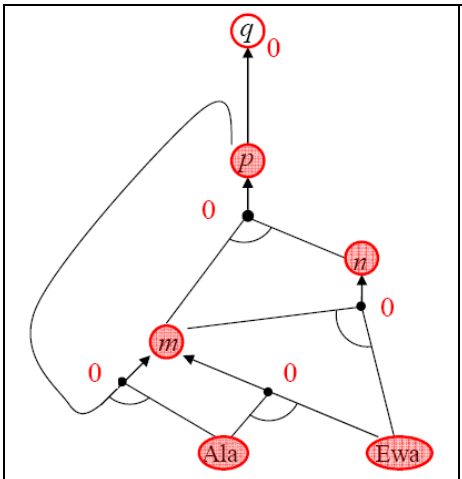
Ewa

Odpowiada im reprezentacja graficzna w postaci grafu I-LUB (rys. 2.5). Każdy węzeł typu „I" posiada etykietę – odpowiada ona liczbie warunków w poprzedniku reguły pozostałych jeszcze do spełnienia. Kolejno wykonywane kroki wnioskowania w przód ilustruje rys. 2.6.

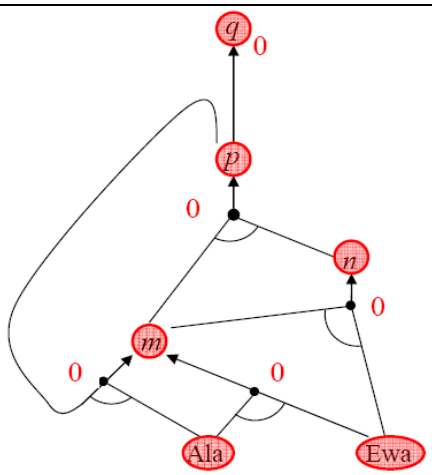


Rys. 2.5: Graf I-LUB dla reprezentacji zbioru klauzul Horna.





(g) Próbuj ponownie dodać *m*



(h) Dodaj *q*

Rys. 2.6: Przykład wnioskowania „w przód”

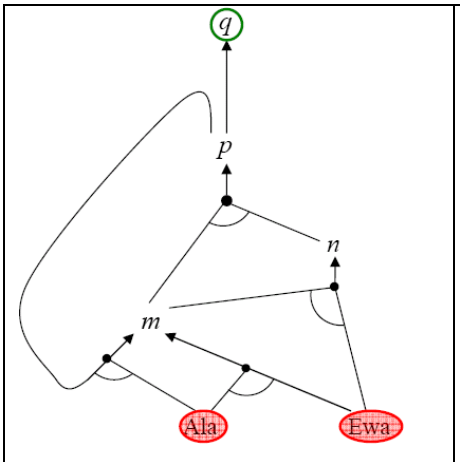
2.6.2. Wnioskowanie wstecz

Idea procedury wnioskowania „wstecz” w rachunku zdań wygląda następująco:

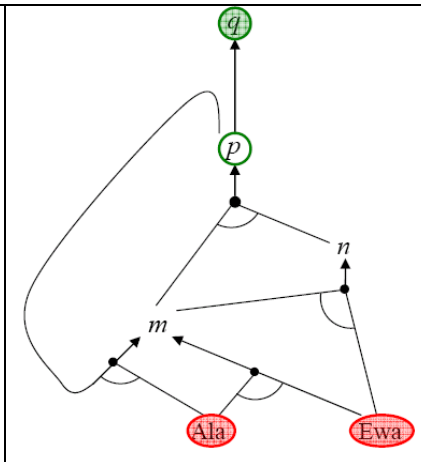
1. Funkcja rozpoczyna od zdania zapytania (celu) *q*.
2. Aby sprawdzić prawdziwość *q* procedura sprawdza, czy *q* już występuje w KB a jeśli nie, to sprawdza czy istnieje przynajmniej jedna implikacja wyprowadzająca zdanie *q*. Jeśli tak, to literały stanowiące warunek tej implikacji stają się „pod-celami” i rekurencyjne badana będzie ich prawdziwość z punktu widzenia aktualnego modelu KB, podobnie jak poprzednio główny cel.
3. Unikanie zapętleń: procedura sprawdza, czy aktualny „pod-cel” nie znajduje się już na stosie wygenerowanych „pod-celów”.
4. Unikanie powielania przejść: sprawdza, czy nowy „pod-cel” został już sprawdzony i pokazano to, czy jest prawdziwy lub fałszywy.

Przykład wnioskowania „wstecz”.

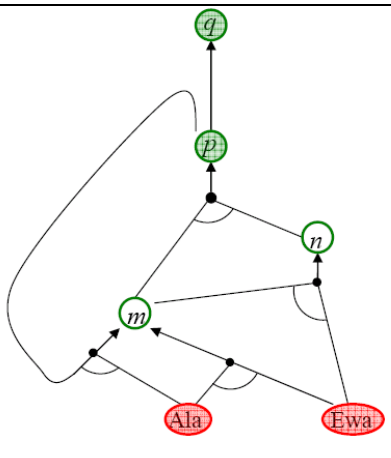
Załóżmy, że baza danych zawiera zdania z poprzedniego przykładu (rys. 2.5). Wśród nich występują też 2 fakty: „Ala” i „Ewa”. Niech zdanie zapytania to „*q*”. Kolejne kroki w procesie wnioskowania wstecz dla tej sytuacji ilustruje rys. 2.7.



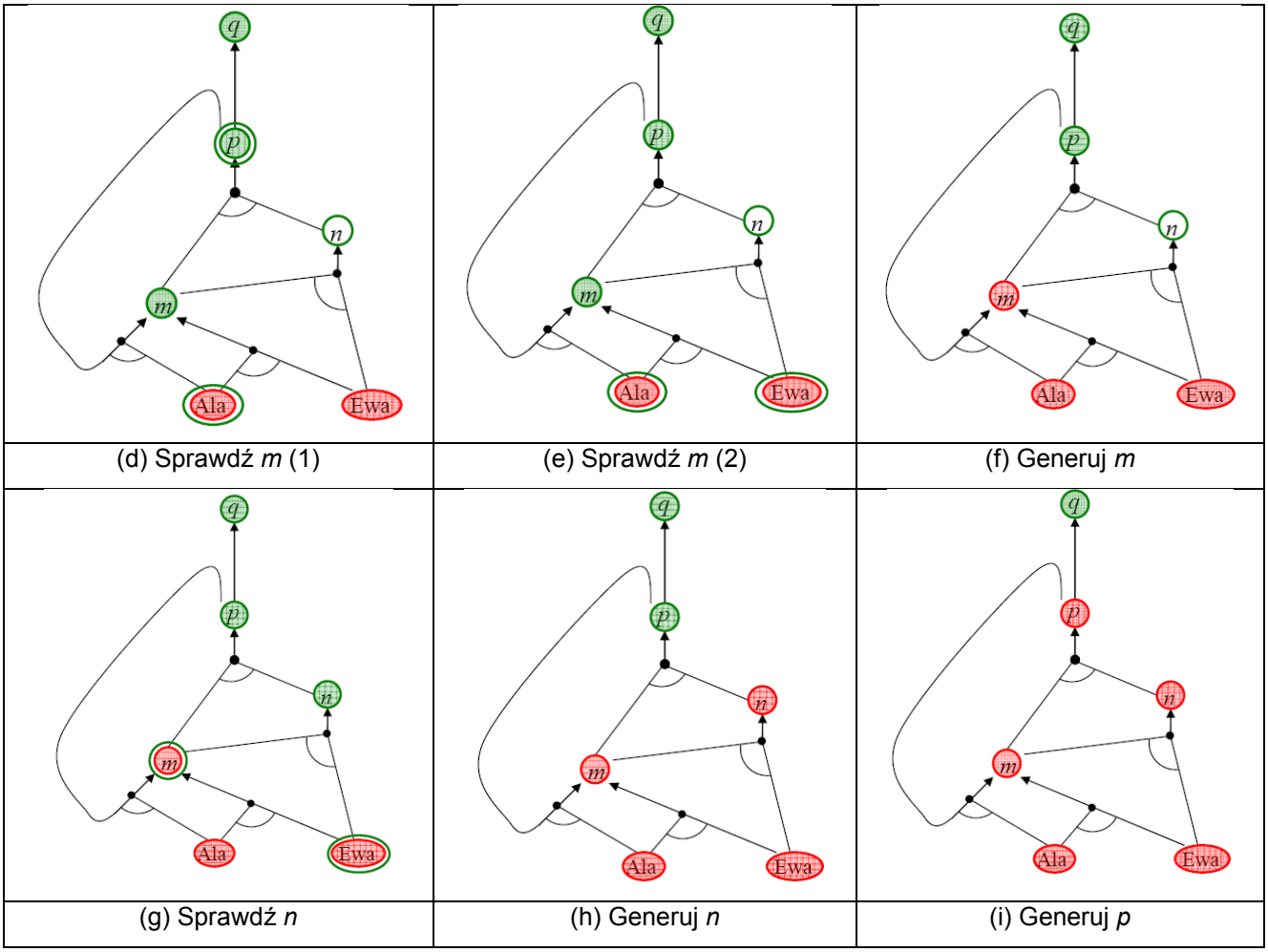
(a) Cel: *q*



(b) Sprawdź: *q*



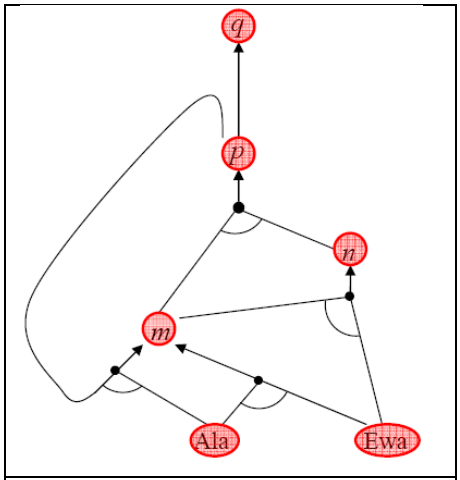
(c) Sprawdź *p*



(d) Sprawdź *m* (1) | (e) Sprawdź *m* (2) | (f) Generuj *m*

(g) Sprawdź *n* | (h) Generuj *n* | (i) Generuj *p*

Rys. 2.7: Przykład wnioskowania „wstecz” (kroki a-i)



(j) Generuj *q*

Rys. 2.7 (c.d.): Przykład wnioskowania „wstecz” – krok końcowy

Procedura wnioskowania w przód jest sterowana danymi – odpowiada to automatycznemu, „nieświadomemu” przetwarzaniu danych. Np. rozpoznawanie obiektów, rutynowe decyzje. Może wykonywać „nadmiarową” pracę, która nie zmierza bezpośrednio do celu.

Procedura wnioskowania wstecz jest sterowana celem – jest to odpowiednie dla rozwiązywania zadanego problemu. Np. dostarczenie odpowiedzi na pytanie: „Gdzie są moje klucze?” Złożoność procedury regresywnej może w praktyce być poniżej liniowej względem rozmiaru KB.

2.7. Własności zmienne w czasie

Prostota rachunku zdań sprawia, że posiada on małą siłę wyrazu – każdy kolejny fakt musi być reprezentowany oddzielnym symbolem języka. Dla przykładu zastanówmy się jak reprezentować w rachunku zdań akcje agenta „*świata Wumpusa*” [4] wykonywane w określonym czasie. Wprowadzamy symbole $L_{i,j}$ dla oznaczenia, że agent działający w 2-wymiarowym środowisku znajduje się w kratce o współrzędnych $[i,j]$. Wtedy możliwym akcjom odpowiadałyby zdania w rodzaju:

$$L_{1,1} \wedge ZwróconyWPrawo \wedge RuchWPrzód \Rightarrow L_{2,1}$$

Jednak to nie prowadzi do prawidłowego wnioskowania. Po wykonaniu akcji oba zdania $L_{1,1}$ i $L_{2,1}$ będą w bazie danych uważane za prawidłowe, tymczasem już tak nie jest, gdyż świat zmienia się wraz z upływem czasu. Jak reprezentować te zmiany w rachunku zdań? Jedynym sposobem jest odpowiednie indeksowanie symboli. Np.:

$$L^1_{1,1} \wedge ZwróconyWPrawo^1 \wedge RuchWPrzód^1 \Rightarrow L^2_{2,1}$$

$$ZwróconyWPrawo^1 \wedge ObrótWLewo^1 \Rightarrow ZwróconyWGórę^2$$

Powyższy przykład ilustruje ograniczenia w sile wyrazu właściwe dla rachunku zdań. Baza wiedzy musi zawierać liczne formuły o podobnej formie różniące się „indeksami”, gdzie każdy zbiór „indeksów” identyfikuje inne „fizyczne” miejsce świata lub czas (np. kratkę w „świecie Wumpusa” w pewnej chwili czasu). Nie ma możliwości wyrażenia w zwarty sposób wspólnej własności wszystkich „fizycznych” miejsc.

Podobnie jest z reprezentacją możliwych akcji agenta - musimy wprowadzić osobne symbole i zdania dla każdej chwili czasu t i każdego miejsca $[x,y]$. W „świecie Wumpusa” dla każdego kierunku, każdej kratki i czasu musiałaby istnieć formuły o postaci

$$L^t_{x,y} \wedge ZwróconyWPrawo^t \wedge RuchWPrzód^t \Rightarrow L^{t+1}_{x+1,y}$$

Efektem jest „eksplozja” liczby zdań w bazie wiedzy przy rosnącym rozmiarze świata i liczbie wykonanych akcji.

2.8. Pytania

1. Omówić **elementy składni** rachunku zdań.
2. Wyjaśnić **semantykę** rachunku zdań.
3. Przedstawić dwa twierdzenia o **dedukcji** (poprawnym wnioskowaniu)
4. Przedstawić typowe **reguły** wnioskowania.
5. Jakie są postacie **normalne** zdań ?
6. Omówić wnioskowanie przez **rezolucję**.
7. Omówić wnioskowanie „**w przód**” i „**wstecz**”.
8. Przedstawić problem własności **zmiennych w czasie** i **wspólnych własności miejsc**.

2.9. Zadania

Zad. 2.1

Wnioskowanie z tablicą prawdy w rachunku zdań dla *świata Wumpusa.*

Niech formuła $J_{i,j}$ będzie prawdziwa, gdy jest jama w kratce [i, j]. Niech formuła $W_{i,j}$ będzie prawdziwa, gdy jest wiatr w kratce [i, j].

Jak wiemy, każdy *model* KB wyznacza wartość „*prawda*" dla zdań zawartych w KB. Przyjmijmy założenie dla *stanu początkowego* -

Zdanie R_1: $\neg J_{11}$ („nie ma jamy w kratce [1,1]")

Zasada budowy świata: dla każdej kratki zachodzi „agent odczuwa wiatr gdy w sąsiedniej kratce jest jama". Możemy to wyrazić dla dwóch pierwszych kratek jako:

Zdanie R_2: $W_{11} \Leftrightarrow (J_{12} \vee J_{21})$

Zdanie R_3: $W_{21} \Leftrightarrow (J_{11} \vee J_{22} \vee J_{31})$

Niech po dwóch obserwacjach (wykonanych w kratkach [1,1] i [2,1]) baza wiedzy zawiera też zdania:

Zdanie R_4: $\neg W_{11}$ („agent nie odczuwa wiatru w [1,1]")

Zdanie R_5: $W_{2,1}$ („agent odczuwa wiatr w [2,1]").

Metodą tablicy prawdy wyznaczyć (przeliczyć) modele aktualnej bazy wiedzy ograniczonej do zdań $R_1 - R_5$.

Zad. 2.2

Zdefiniować w rachunku zdań aksjomaty ukrytych własności dla *świata Wumpusa.*
Uwaga: ***Aksjomaty „ukrytych własności"*** *łączą widoczne obserwacje z ukrytymi przyczynami (zasadami budowy świata).*

Zad. 2.3

Załóżmy, że KB agenta *świata Wumpusa* zawiera zdania zdefiniowane w rozwiązaniu zad. 2.2 do których dodane zostały zdania obserwacji po wizytacji 2 dalszych kratek [1,2] i [2,1]:

$\neg Z_{11}, \neg W_{11}, \neg Z_{21}, W_{21}, Z_{12}, \neg W_{12}$.

Podać, jakie zdania **wynikają** z tak rozszerzonej bazy wiedzy.

Zad. 2.4

Które z podanych niżej reguł wnioskowania są poprawne? Uzasadnić odpowiedź.

1) $\dfrac{\alpha \to \beta, \beta \to \gamma}{\alpha \to \gamma}$

2) $\dfrac{\alpha \to \beta, \beta \to \gamma, \alpha}{\gamma}$

3) $\dfrac{\alpha \vee \beta, \alpha \vee \neg\beta}{\alpha}$

4) $\dfrac{\alpha \to \beta}{\neg\beta \to \neg\alpha}$

5) $\dfrac{\alpha \to \beta}{\neg\alpha \to \neg\beta}$

6) $\dfrac{\alpha \to (\beta \to \gamma)}{\beta \to (\alpha \to \gamma)}$

Zad. 2.5

Przekształcić poniższe zdanie do postaci normalnej dla wnioskowania **z regułą rezolucji**:

$$A1 \Leftrightarrow (L1 \vee L2).$$

Wyjaśnić realizowane przekształcenia.

Zad. 2.6

Zaproponować sposób reprezentacji własności zmiennych w czasie na gruncie rachunku zdań.

3. Logika predykatów – reprezentacja wiedzy

3.1. Składnia języka logiki pierwszego rzędu

Podczas gdy rachunek zdań zakłada, że świat składa się z faktów, **logika pierwszego rzędu** (**logika predykatów**), podobnie jak język naturalny, pozwala specyfikować te fakty znacznie dokładniej.

3.1.1. Kategorie symboli języka

W logice predykatów zakładamy, że w świecie występują:

1. Obiekty

   Np. osoby, domy, liczby, kolory, gry, wojny, …

2. Relacje

   Np. jest czerwony, jest okrągły, jest liczbą pierwszą, jest bratem, większy niż, jest częścią, jest pomiędzy, …

3. Funkcje

   Np. jego ojciec, jego najlepszy przyjaciel, o jeden więcej, suma, …

Pozwala to wyrazić fakty jako relacje zachodzące na obiektach wyznaczanych przez funkcje. Stąd składnia języka predykatów obejmuje następujące zasadnicze elementy:

- Stałe np. *KrólJan, 2, PW*,...
- Symbole predykatów np. *Brat, >,...*
- Symbole funkcji np. *Sqrt, LewaNoga*,...
- Zmienne *x, y, a, b*,...
- Negacja i spójniki $\neg, \Rightarrow, \wedge, \vee, \Leftrightarrow$
- Predykat równości =
- Kwantyfikatory (dla wyrażenia własności zbioru obiektów) $\forall, \exists$

3.1.2. Wyrażenia – termy i formuły

Z powyższych elementów tworzone są wyrażenia języka: termy i formuły. **Termy** wskazują na obiekty i są ogólnej postaci funkcji, stałej lub zmiennej:

$$funkcja\ (term_1,...,term_n)$$

lub *stała* lub *zmienna*

Formuła atomowa to wyrażenie zbudowane na pojedynczym predykacie, tzn. o ogólnej postaci:

$$predykat\ (term_1,...,term_n)$$

lub

$$term_1 = term_2$$

Predykat „=” („równość”) został wyróżniony dlatego, gdyż we wszystkich zastosowaniach języka powinien posiadać podobne znaczenie.

Przykłady:

Term: *Brat*(*Jan*);

Formuła atomowa: *>(Długość(LewaNoga(Ryszard)), Długość(LewaNoga(Jan)))*

Formuły złożone powstają z połączenia formuł atomowych spójnikami z możliwością wykorzystania kwantyfikatorów i negacji:

$$\neg S, \quad S_1 \wedge S_2, \quad S_1 \vee S_2, \quad S_1 \Rightarrow S_2, \quad S_1 \Leftrightarrow S_2,$$

Przykłady formuł:

Rodzeństwo(*Jan, Ryszard*) $\Rightarrow$ *Rodzeństwo*(*Ryszard, Jan*)

>(1,2) $\vee$ ≤ (1,2)

>(1,2) $\wedge$ $\neg$ >(1,2)

3.1.3. Kwantyfikatory

Uniwersalny kwantyfikator jest ogólnej postaci:

$\forall$<*zmienne*> <formuły>

Np. Zdanie „*Każda osoba studiująca na PW jest inteligentna*” zapiszemy z użyciem kwantyfikatora jako: $\forall x$ (*Studiuje*(*x*, PW) $\Rightarrow$ *Inteligentna*(*x*)).

Wartościowanie formuły z kwantyfikatorem ogólnym: formuła $\forall$x *P* jest prawdziwa w modelu *M* wtw. gdy *P* jest prawdziwe dla *x* wartościowanego dowolnym obiektem w tym modelu. W przybliżeniu jest to równoważne koniunkcji wszystkich możliwych wartościowań *P:*

(*Studiuje*(Jan, PW) $\Rightarrow$ *Inteligentna*(Jan)) $\wedge$ (*Studiuje*(Ryszard, PW) $\Rightarrow$ *Inteligentna*(Ryszard)) $\wedge$ (*Studiuje*(PW, PW) $\Rightarrow$ *Inteligentna*(PW)) $\wedge$ ...

Egzystencjalny kwantyfikator jest ogólnej postaci

$\exists$<*zmienne*> <formuły>

Np. Zdanie „*Ktoś spośród osób studiujących na PW jest inteligentny*” zapiszemy jako:

$$\exists x\ (Studiuje(\mathrm{x}, \mathrm{PW}) \wedge Inteligentna(\mathrm{x}))$$

Formuła $\exists x$ *P* jest prawdziwa w modelu *M* wtw. gdy *P* jest prawdziwe dla *x* wartościowanego jakimś obiektem modelu. W przybliżeniu jest to równoważne alternatywie różnych wartościowań *P.* Np.

$$(Studiuje(\mathrm{Jan},\ \mathrm{PW}) \wedge Inteligentna(\mathrm{Jan})) \vee (Studiuje(\mathrm{Ryszard},\ \mathrm{PW}) \wedge Inteligentna(\mathrm{Ryszard})) \vee (Studiuje(\mathrm{PW}, \mathrm{PW}) \wedge Inteligentna(\mathrm{PW})) \vee ...$$

Własności kwantyfikatorów:

- $\forall$x $\forall$y jest taka sama jak $\forall$y $\forall$x
- $\exists$x $\exists$y jest taka sama jak $\exists$y $\exists$x
- $\exists$x $\forall$y nie jest taka sama jak $\forall$y $\exists$x

Np. formuła, $\exists$x $\forall$y *Kocha*(x, y), oznaczająca “*Istnieje osoba, która kocha wszystkich w świecie*”, nie jest tożsama z formułą, $\forall$y $\exists$x *Kocha*(x,y), oznaczającą “*Każdy na świecie jest kochany przez przynajmniej jedną osobę*”.

Dualność kwantyfikatorów: każdy kwantyfikator może być wyrażony przez drugi z nich. Np.:

$$\forall x\ Lubi(\mathrm{x}, \mathrm{lody}) \quad \equiv \quad \neg\exists x\ \neg Lubi(\mathrm{x}, \mathrm{lody})$$

$\exists$x *Lubi*(x, brokuły) $\equiv$ $\neg\forall$x $\neg$*Lubi*(x, brokuły)

3.2. Semantyka języka predykatów

3.2.1. Model i wartościowanie zmiennych

W przypadku języka predykatów znaczenie formuły określamy ze względu na: ustaloną dziedzinę *D,* funkcję interpretacji *m* i wartościowanie a. Model zbioru formuł nadal jest postaci, *M* = [*D*, *m*], ale aby wyznaczyć wartość formuły należy tez wartościować jej zmienne, a to wymaga podania funkcji wartościowania *a*.

Dziedzina *D* zawiera obiekty (elementy dziedziny) i relacje oraz funkcje pomiędzy nimi.

Funkcja interpretacji *m* przyporządkowuje:

- symbolom stałych → obiekty, czyli elementy dziedziny

- *n*-arg. symbolom predykatów → *n*-argumentowe relacje określone na D^n

- n-arg. symbolom funkcyjnym → funkcje z D^n na D

Formuła atomowa o postaci *predykat*(*term*$_1$,...,*term*$_n$) *jest prawdziwa* (*True*) w interpretacji *m* wtw. gdy obiekty wyznaczone przez *term*$_1$,...,*term*$_n$ spełniają relację referowaną przez *predykat.*

W jaki sposób wyznaczamy obiekt będący wartością termu? Przy interpretowaniu formuły w danym modelu nadawanie wartości zmiennym tej formuły określa się mianem **wartościowania**. Wartościowanie zmiennych a w zadanym modelu M to odwzorowanie symboli zmiennych na elementy dziedziny.

Oznaczmy przez $V_a^M(t)$ wartość termu *t* w modelu $M = [D, m]$ względem wartościowania a. Wartość stałej lub zmiennej obliczamy jako:

- $V_a^M(x) = a(x)$, gdzie x jest zmienną.
- $V_a^M(C) = m(C)$, gdzie C jest stałą.

Wartość termu $f(t_1,...,t_n)$ wynosi:

- $V_a^M(f(t_1,...,t_n)) = m(f)(V_a^M(t_1)\ ,...,\ V_a^M(t_n)\ )$.

Oznaczmy przez $V_a^M(\tau)$ wartość formuły τ w modelu M, gdzie $M = [D, m]$, względem wartościowania a.

- Dla predykatu P: $V_a^M(P(t_1,...,t_n)) = m(P)(V_a^M(t_1)\ ,...,\ V_a^M(t_n)\ )$.
- Dla równości: $V_a^M(t_1 = t_2)) = (V_a^M(t_1) = V_a^M(t_2)\ )$.
- Dla formuł złożonych:

$$V_a^M(B \wedge C) = V_a^M(B) \wedge V_a^M(C)$$
$$V_a^M(B \vee C) = V_a^M(B) \vee V_a^M(C)$$
$$V_a^M(B \Rightarrow C) = V_a^M(B) \Rightarrow V_a^M(C)$$
$$V_a^M(B \Leftrightarrow C) = V_a^M(B) \Leftrightarrow V_a^M(C)$$

- Dla kwantyfikatorów:

$$V_a^M(\forall x\ B) = \min_{d \in D}[V_{a(x \leftarrow d)}^M(B)]$$
$$V_a^M(\exists x\ B) = \max_{d \in D}[V_{a(x \leftarrow d)}^M(B)]$$

gdzie *min*(True, False) wynosi False, a *max*(True, False) wynosi True.

Niech $a(x \leftarrow d)$ oznacza wartościowanie identyczne z a dla wszystkich zmiennych poza x, w którym zmiennej x nadawana jest wartość d. Czyli $a(x \leftarrow d)$ różni się od a co najwyżej wartościowaniem zmiennej x.

3.2.2. Rodzaje formuł

W dalszym ciągu badamy znaczenie formuł względem ustalonej dziedziny D.

Powiemy, że formuła A jest **spełnialna** jeśli istnieje interpretacja m i wartościowanie a przy których A jest spełniona (tzn. ma wartość $True$).

A jeśli jej prawdziwość nie zależy od wartościowania? Formuła A jest **prawdziwa** w interpretacji m jeśli ma wartość $True$ w tej interpretacji przy każdym wartościowaniu a. Mówimy wtedy, że $[D, m]$ jest modelem formuły A.

A co jeśli jej prawdziwość nie zależy od funkcji interpretacji? Mówimy, że formuła A jest **tautologią** jeśli jest *prawdziwa* przy każdej interpretacji m i wartościowaniu a (tzn. jest zawsze prawdziwa dla zadanej dziedziny D).

Teoria jest to zbiór formuł tworzony dla określonej dziedziny. Teoria jest **niesprzeczna** jeśli istnieją: interpretacja i funkcja wartościująca, dla których prawdziwe są wszystkie formuły teorii.

Aksjomaty teorii to formuły (uznane za) prawdziwe niezależnie od wartościowania. Interesują nas teorie, których formuły dadzą się wyprowadzić w procesie wnioskowania używając aksjomatów i podzbioru formuł początkowej bazy wiedzy jako „punktów startowych" wnioskowania.

Np. aksjomaty w teorii (w świecie) „Osoby spokrewnione":

„Bracia są rodzeństwem" : $\forall x,y\ Brat(x,y) \Leftrightarrow Rodzeństwo(x,y)$

„Matka jest rodzicem i kobietą" : $\forall m,c\ Matka(c) = m \Leftrightarrow (Kobieta(m) \wedge Rodzic(m,c))$

"Rodzeństwo" jest symetryczną relacją : $\forall x,y\ Rodzeństwo(x,y) \Leftrightarrow Rodzeństwo(y,x)$

Równość jest wyróżnionym predykatem, który w każdej interpretacji ma to samo znaczenie – oznacza on relację identyczności, tzn.

$(term_1 = term_2)$ jest *prawdziwe* wtw. gdy $term_1$ i $term_2$ referują ten sam obiekt.

Z predykatu równości korzystamy często opisując własności funkcji lub definiując predykaty w terminach innych funkcji lub relacji, poprzez wprowadzenie wymogu równości lub różności obiektów. Np. poniższa definicja predykatu *Rodzeństwo* korzysta z równości termów i prawdziwości relacji *Rodzic*:

$\forall x,y\ Rodzeństwo(x,y) \Leftrightarrow$
$[\neg(x = y) \wedge \exists m,f \neg(m = f) \wedge Rodzic(m,x) \wedge Rodzic(f,x) \wedge Rodzic(m,y) \wedge Rodzic(f,y)]$

3.3. Przekształcanie formuł

3.3.1. Postać predykatowa

Czy możemy dla wnioskowania formuł stosować te same reguły wnioskowania co dla rachunku zdań? Tak, jeśli sprowadzimy formuły do tzw. postaci *predykatowej* (zdaniowej), tzn. wyeliminujemy zmienne i termy.

Załóżmy, że KB składa się z formuły złożonej skwantyfikowanej uniwersalnie,

$\forall x\ Król(x) \wedge Chciwy(x) \Rightarrow Zły(x)$,

i z literałów (faktów): Król(Jan) , Chciwy(Jan) , Brat(Ryszard, Jan).

Wartościując formułę uniwersalną wszystkimi możliwymi (znanymi) obiektami otrzymamy KB:

Król(Jan) $\wedge$ Chciwy(Jan) $\Rightarrow$ Zły(Jan)

Król(Ryszard) $\wedge$ Chciwy(Ryszard) $\Rightarrow$ Zły(Ryszard)

Król(Jan) , Chciwy(Jan) , Brat(Ryszard, Jan)

Tak powstała KB nie zawiera zmiennych. Symbolami *predykatowymi* o charakterze zdań są:

Król(Jan), Chciwy(Jan), Zły(Jan), Król(Ryszard), itd.

Można je traktować jak symbole zdaniowe i stosować wnioskowanie w rachunku zdań. Prawdziwe jest twierdzenie:

„Każda KB w L1R może zostać przekształcona do postaci *predykatowej*, zachowującej własność wynikania, czyli: bazowa formuła wynika z nowej KB wtw. gdy wynika z oryginalnej KB."

Jednak powstaje zasadniczy problem związany z potencjalnie nieskończoną liczbą symboli predykatowych, które mogą powstać ze skończonej liczby formuł. Z powodu istnienia symboli funkcji istnieje nieskończenie wiele termów. Np. Ojciec(Jan), …, Ojciec(Ojciec(Ojciec(Jan))) …

Dlatego też w logice predykatów stosuje się **uogólnione reguły wnioskowania**, tzn. reguły znane nam z rachunku zdań zostają uogólnione dzięki uwzględnieniu podstawień pod zmienne i mechanizmu unifikacji formuł (uzgadniania zmiennych).

3.3.2. Podstawienie pod zmienne

Rozszerzymy teraz reguły wnioskowania z rachunku zdań do odpowiednich postaci wymaganych przez logikę predykatów.

Przez **podstawienie** rozumiemy zbór par postaci $\{x_1/t_1,...,x_n/t_n\}$, gdzie $x_1,...,x_n$ są różnymi zmiennymi, natomiast, $t_1,...,t_n$, są termami. Dopuszczamy podstawienie puste ε.

Niech θ będzie podstawieniem i niech α będzie wyrażeniem (tzn. formułą lub termem). Przez **SUBST**(θ , α) oznaczamy wyrażenie powstałe w wyniku jednoczesnego zastąpienia wszystkich wolnych wystąpień zmiennych, $x_1,...,x_n$, termami, $t_1,...,t_n$.

Np. jeśli $\theta = \{x/\text{Jan}, y/\text{Ewa}\}$, to

$$\text{SUBST}(\theta, Lubi(x, y)) = Lubi(\text{Jan}, \text{Ewa}).$$

Przypomnijmy, że literał jest to formuła atomowa lub negacja formuły atomowej, a zdanie - formuła bez zmiennych.

Przykład

Stosując predykatowanie generujemy zwykle wiele niepotrzebnych formuł.

Np. dla zbioru $\forall x$ *Król*(x) $\wedge$ *Chciwy*(x) $\Rightarrow$ *Zły*(x) , *Król*(*Jan*) , $\forall y$ *Chciwy*(y) , *Brat*(*Ryszard, Jan*)

wydaje się być oczywiste, że jest model w którym zachodzi *Zło*(*Jan*), ale predykatowanie produkuje również szereg innych faktów, takich jak: *Chciwy*(*Ryszard*), które nie mają modelu.

W ogólności, gdy dziedzina liczy p k-argumentowych predykatów i n obiektow, istnieje $p \cdot n^k$ możliwych wartościowań. Łatwo unikniemy tego nadmiaru formuł, jeśli w jednym kroku wnioskowania znajdziemy podstawienie θ takie, dla którego formuły *Król*(x) i *Chciwy*(x) są równoważne z istniejącymi *Król*(*Jan*) i *Chciwy*(y). Pasuje nam tu: θ = {x/*Jan*, y/*Jan*}.

3.3.3. Uzgadnianie zmiennych i unifikacja formuł

Obecność symboli zmiennych w logice predykatów powoduje to, że nie możemy poprzestać na prostym wymaganiu identyczności formuł tworzących poprzednik reguły wnioskowania. Stosujemy łagodniejsze wymaganie takie, że pierwsza i druga formuła wejściowa (tworzące poprzednik reguły wnioskowania) są **unifikowalne**, czyli mogą zostać sprowadzone do postaci identycznej przez

zastosowanie odpowiednich podstawień dla zmiennych - przypisanie im obiektów lub pewnych termów. W istocie mamy do czynienia z dwoma przypadkami **uzgadniania zmiennych**:

1. Ujednolicanie zmiennych – wtedy, gdy dwie formuły różnią się jedynie tym, że w odpowiednich miejscach występują w nich konsekwentnie inne symbole zmiennych,

2. Uszczegółowienie – wtedy, gdy w miejscach, gdzie w jednej z nich występuje pewien symbol zmiennej (związany kwantyfikatorem ogólnym), w drugiej konsekwentnie występuje pewien term nie będący zmienną (stała albo zastosowanie symbolu funkcyjnego).

Mówimy, że podstawienie θ jest unifikatorem wyrażeń, $E_1,...,E_n$, jeśli

$$\text{SUBST}(\theta, E_1) = ... = \text{SUBST}(\theta, E_n).$$

Np. podstawienie $\{x/A\}$ jest unifikatorem wyrażeń $P(x), P(A)$. Ale wyrażenia $P(x), Q(b)$ nie są unifikowalne.

Reguły wnioskowania w logice predykatów stosują **algorytm unifikacji** (nazwijmy go $\text{UNIFY}(p,q)$), który zwraca takie podstawienie (tzw. **najogólniejszy unifikator**) dla literałów p i q, które czyni je *identycznymi*, lub zwraca *błąd*, jeśli podstawienie nie istnieje. Czym jest najogólniejszy unifikator? **Złożeniem** podstawień θ_1, θ_2 jest takie podstawienie (oznaczmy je przez $\text{COMPOSE}(\theta_1, \theta_2)$), które polega na wykonaniu po kolei obu podstawień, czyli:

$$\text{SUBST}(\text{COMPOSE}(\theta_1, \theta_2), E) = \text{SUBST}(\theta_2, \text{SUBST}(\theta_1, E)).$$

Podstawienie θ jest **najogólniejszym** unifikatorem dla $\{E_1,..., E_n\}$, jeśli dla każdego unifikatora γ dla tych wyrażeń istnieje podstawienie λ takie, że: $\gamma = \text{COMPOSE}(\theta, \lambda)$. Innymi słowy najogólniejszy unifikator zawiera jedynie to co niezbędne dla unifikacji dwóch literałów, na których działa. Ważną dodatkową obserwacją jest to, że unifikowalne wyrażenia posiadają dokładnie jeden najogólniejszy unifikator.

Przykład. Wyrażenia $P(\text{Jan}, x), P(y, z)$ posiadają nieskończenie wiele unifikatorów. Niektóre z nich to: $\{y/\text{Jan}, x/z\}$, $\{y/\text{Jan}, x/z, w/\text{Fred}\}$, $\{y/\text{Jan}, x/\text{Jan}, z/\text{Jan}\}$. Najogólniejszym unifikatorem jest:

$$\{y/\text{Jan}, x/z\}$$

Problem uzgadniania zmiennych w wyrażeniach, $E_1,..., E_n$, polega najpierw na zbadaniu, czy dane wyrażenia są unifikowalne, a jeśli tak to należy znaleźć ich najogólniejszy unifikator.

Problem unifikacji jest rozstrzygalny. Wynika to ze skończonej liczby zmiennych w literałach.

Przykład unifikacji literałów:

$p = \text{Zna}(\text{Jan}, x);$	$q = \text{Zna}(\text{Jan}, \text{Ewa});$	$\text{UNIFY}(p,q) = \{x/\text{Ewa}\}$
$p = \text{Zna}(\text{Jan}, x)$	$q = \text{Zna}(y, \text{Ewa})$	$\{x/\text{Ewa}, y/\text{Jan}\}$
$p = \text{Zna}(\text{Jan}, x)$	$q = \text{Zna}(y, \text{Matka}(y))$	$\{y/\text{Jan}, x/\text{Matka}(\text{Jan})\}$
$p = \text{Zna}(\text{Jan}, x)$	$q = \text{Zna}(x, \text{Ewa})$	*błąd*

3.3.4. Standaryzacja rozłączna

Możemy otrzymać równoważne warianty zadanej formuły zmieniając nazwy zmiennych tej formuły (*przemianowanie zmiennych*). Powiemy, że formuła A jest **wariantem** formuły B, jeśli A można otrzymać z B zastępując niektóre zmienne w B nowymi zmiennymi nie występującymi w B.

Np. $Lubi(x, \text{Jan})$ jest wariantem formuły $Lubi(y,\text{Jan})$.

W zasadzie każda formuła w bazie wiedzy może zostać zastąpiona swoim wariantem (przemianowana) i otrzymamy bazę wiedzy równoważną z poprzednią postacią. Jednak dlaczego należy przemianowywać zmienne w formule dodawanej do bazy wiedzy? Jest to motywowane potrzebą uniknięcia błędów unifikacji, które powodowane są przez powtarzanie się nazw zmiennych, w sytuacji, gdy w oczywisty sposób nie musza się one odnosić do tego samego obiektu.

Np. załóżmy, że w istnieją dwie formuły: $Zna(x, Ewa)$, $Zna(Jan, x) \Rightarrow Nienawidzi(Jan, x)$. Powinniśmy móc wywnioskować na podstawie tej pary, stosując ogólną regułę *Modus Ponens*, że zachodzi

Nienawidzi(*Jan, Ewa*). W praktyce nie jest to jednak możliwe, gdyż formuły, *Zna*(*Jan, x*) i *Zna*(*x, Ewa*), nie są unifikowalne. Dopiero potem, jak przemianujemy jedną ze zmiennych x (np. formułę *Zna*(*x, Ewa*), zamienimy na, *Zna*(*y, Ewa*)) to będziemy mogli wyprowadzić *Nienawidzi*(*Jan, Ewa*).

Proces przemianowania zmiennych występujących w formułach, tak, aby każda formuła posiadała unikalne zmienne nazywamy **standaryzacją rozłączną** formuł. Możemy założyć, że formuły w KB nie zawierają wspólnych zmiennych, gdyż zostały wcześniej poddane standaryzacji rozłącznej.

3.3.5. Eliminacja kwantyfikatorów

W bazie wiedzy występują formuły pozbawione kwantyfikatorów. Formuła definiowana przez eksperta, w której zmienne związane są kwantyfikatorami, może zostać przekształcona do równoważnej postaci formuły pozbawionej kwantyfikatorów.

Zasada eliminacji kwantyfikatorów:

1. Standaryzacja rozłączna zmiennych – każdy kwantyfikator wiąże unikalną zmienną.
2. Skolemizacja – eliminacja kwantyfikatorów szczegółowych (egzystencjalnych).
3. Eliminacja kwantyfikatorów uniwersalnych.

Eliminacja kwantyfikatora szczegółowego

Dla każdej formuły α, zmiennej v, i symbolu stałej K, który nie występuje nigdzie indziej w bazie wiedzy, zachodzi reguła wnioskowania o postaci:

$$\frac{\exists v\ \alpha}{SUBST(\{v/K\},\alpha)}$$

Np. z formuły, ∃x Korona(x) ∧ NaGłowie(x, Jan), wynika:

Korona(C_1) ∧ NaGłowie(C_1,Jan),

pod warunkiem, że C_1 jest nowym symbolem stałej, zwanej stałą Skolema.

Jeśli kwantyfikator egzystencjalny formuły poprzedzony jest kwantyfikatorem uniwersalnym zmiennej x to za v podstawiamy unikalny symbol funkcji, zwanej funkcją Skolema, o parametrze x:

$$\frac{\forall x\ \exists v\ \alpha}{SUBST(\{v/F(x)\},\alpha)}$$

Eliminacja kwantyfikatora uniwersalnego

Każde wartościowanie formuły związanej uniwersalnym kwantyfikatorem wynika z tej formuły :

$$\frac{\forall v\ \alpha}{SUBST(\{v/g\},\alpha)}$$

dla każdej zmiennej v i bazowego termu g (co najwyżej jedna funkcja w termie).

Np. z ∀x Król(x) ∧ Chciwy(x) ⇒ Zły(x) wynika:

Król(Jan) ∧ Chciwy(Jan) ⇒ Zły(Jan)

Król(Ryszard) ∧ Chciwy(Ryszard) ⇒ Zły(Ryszard)

Król(Ojciec(Jan)) ∧ Chciwy(Ojciec(Jan)) ⇒ Zły(Ojciec(Jan))

Powyższa reguła oznacza w praktyce, że po wykonaniu standaryzacji rozłącznej zmiennych względem kwantyfikatorów w formule i po eliminacji kwantyfikatorów szczegółowych, można już po prostu opuścić kwantyfikatory uniwersalne w formule. Bowiem zakres oddziaływania takiego kwantyfikatora w formule jest w oczywisty sposób zadany unikalną nazwą jej zmiennej oraz nie istnieje żaden kwantyfikator szczegółowy, który byłby od niego zależny.

3.4. Rachunek sytuacji w logice predykatów

Wprowadzimy teraz podzbiór języka predykatów, określany jako rachunek sytuacji („*situation calculus*") dla modelowania zjawisk i własności zmiennych w czasie.

3.4.1. Przykład

Prosty agent reaktywny będzie posiadał jedynie reguły wiążące bezpośrednio aktualne obserwacje i akcje. Np. reguła wyrażona w logice predykatów, wiążąca aktualną obserwację błysku złota w chwili *t* z wyborem właściwej akcji „Podnieś", może przyjąć postać dwóch formuł języka:

Formuła dla obserwacji (percepcji)

$$\forall_{s,\,w,\,u,\,k,\,t}\ \mathit{Percepcja}([s,\,w,\,\text{Błysk},\,u,\,k],\,t) \Rightarrow \mathit{PrzyZłocie}(t)$$

Formuła dla wyboru akcji (refleks):

$$\forall_{t}\ \mathit{PrzyZłocie}(t) \Rightarrow \mathit{NajlepszaAkcja}(\text{Podnieś},\,t)$$

Dzięki wprowadzeniu zmiennych reprezentacja w logice predykatów jest znacznie efektywniejsza niż w rachunku zdań. Jednak prosty agent reaktywny nie poradzi sobie ze *światem Wumpusa*, gdyż: (1) nigdy nie będzie wiedział na pewno, czy lepiej jest wrócić do kwadratu startowego czy dalej szukać złota, (2) często się zapętli, gdyż nie rozróżnia on czy niesie już złoto czy też jeszcze nie. Przyczyny obu trudności leżą w braku informacji o stanie środowiska.

Agent reaktywny ze stanem reaguje na bieżącą obserwację środowiska w oparciu o jego aktualnie stworzony opis (stan problemu). Baza wiedzy takiego agenta ma charakter dynamiczny – akumuluje ona wszystkie obserwacje w postaci modyfikowalnego stanu problemu.

3.4.2. Wymagane elementy

Rachunek sytuacji („*situation calculus*" - Hayes, McCarthy 1969) jest to pewien sposób opisywania zmian (w czasie) za pomocą języka logiki pierwszego rzędu, czyli do modelowania dynamicznie zmieniającego się świata. Główne jego założenia to:

1. Świat to ciąg sytuacji, z których każda opisuje stan świata w pewnym momencie czasu.
2. Nowa sytuacja powstaje z bieżącej sytuacji w wyniku wykonania akcji przez agenta.

Sytuacje

Zgodnie z tym w rachunku sytuacyjnym każdy symbol predykatu, reprezentujący relację (lub własność) zmieniającą się w czasie, będzie posiadał dodatkowy argument, określający **sytuację**. Sytuacje są to obiekty należące do specyficznej kategorii ***czasu-stanu***.

Przykład

Dla agenta w „świecie Wumpusa" wprowadzimy predykat o 3 argumentach

$$\mathit{Jest}(\text{Agent},\ \mathit{pozycja},\ \mathit{sytuacja}),$$

dla wyrażenia pozycji agenta (kratki położenia i kierunku zorientowania) w określonej sytuacji w 2-wymiarowym świecie. Np. możemy zapisać:

$$\mathit{Jest}(\text{Agent},\,[(1,1),\,90],\,S_0) \wedge \mathit{Jest}(\text{Agent},\,[(1,2),\,90],\,S_1)$$

Funkcja następstwa sytuacji

Kolejne sytuacje (w czasie) nie są jawnie reprezentowane przez predefiniowane obiekty „czasu-stanu" ale są wynikiem poprzedniej sytuacji i wykonanej akcji. W tym celu wprowadzimy funkcję

$$\mathit{Rezultat}(\mathit{akcja},\ \mathit{sytuacja}),$$

która wyznaczy sytuację będącą wynikiem wykonania akcji w zadanej sytuacji poprzedniej.

Np. wyznaczamy nową sytuację bezpośrednio występującą po sytuacji początkowej S_0 jako:

$$S_1 = Rezultat(\text{RuchWPrzód}, S_0) .$$

3.4.3. Aksjomaty efektów akcji

W nowej sytuacji będącej wynikiem wykonanej akcji będą zachodzić nowe relacje (własności) reprezentowane predykatami. Wnioskujemy je dzięki **aksjomatom efektów akcji**, zwykle zadanym w postaci **reguł** (formuły połączone spójnikiem implikacji).

Np. efektem akcji „*Podnieś złoto*" w pozycji „*x*" i sytuacji „*s*" będzie:

$$\forall_{x,s} \quad PrzyZłocie(s) \wedge Jest(Agent, x, s) \Rightarrow TrzymaZłoto(x, Rezultat(\text{Podnieś}, s))$$

Np. efektem akcji „*Puść*" będzie:

$$\forall_{x,s} \neg TrzymaZłoto(\text{x}, Rezultat(\text{Puść}, s))$$

3.4.4. Aksjomaty tła

Aksjomaty **tła** – reprezentują one relacje (własności), które nie zmieniają się po przejściu świata do następnej sytuacji.

Np. *agent trzymający złoto, którego nie upuścił, w następnej sytuacji nadal będzie to złoto trzymał*:

$$\forall a,\mathbf{x},s \; TrzymaZłoto(\mathbf{x},s) \wedge (a \neq \text{Puść}) \Rightarrow TrzymaZłoto(\mathbf{x}, Rezultat(a,s))$$

Np. *jeśli agent nie posiadał złota i go nie podniósł, to nadal go nie posiada*:

$$\forall a,\mathbf{x},s \; \neg TrzymaZłoto(\mathbf{x},s) \wedge (a \neq \text{Podnieś}) \Rightarrow \neg TrzymaZłoto(\mathbf{x}, Rezultat(a,s))$$

3.4.5. Aksjomaty następstwa stanów

Łączymy aksjomaty efektów akcji i aksjomaty tła dla tego samego predykatu w jeden aksjomat **następstwa stanów.** Zebrane są w nim wszystkie warunki zmiany wartości danego predykatu.

Np. dla predykatu *TrzymaZłoto*:

$$\forall a,\mathbf{x},s \; TrzymaZłoto(\mathbf{x}, Rezultat(a,s)) \Leftrightarrow (a = \text{Podnieś}) \vee [\, TrzymaZłoto(\mathbf{x},s) \wedge (a \neq \text{Puść})]$$

3.5. Pytania

1. Omówić elementy składni logiki predykatów.
2. Omówić semantykę logiki predykatów.
3. Przedstawić problem predykatowania formuł.
4. Na czym polegają: podstawienie i uzgadniania zmiennych?
5. Omówić typową kwantyfikację formuł i reguły eliminacji kwantyfikatorów
6. Omówić proces unifikacji formuł.
7. Przedstawić rachunek sytuacji – przeznaczenie, podstawowe elementy.

3.6. Zadania

Zad. 3.1

Jak reprezentować predykaty przedstawiające własności zmienne w czasie na gruncie logiki predykatów?

Zad. 3.2

Zdefiniować w rachunku sytuacyjnym warunki początkowe, aksjomaty miejsc i ukrytych własności miejsc dla „świata Wumpusa”.

Zad. 3.3

Jak reprezentować wybór akcji w rachunku sytuacyjnym dla „świata Wumpusa”?

Zad. 3.4

Zapisać następujące zdania w języku logiki predykatów, wprowadzając niezbędne symbole i ustalając ich interpretację:

1. ojciec każdego człowieka jest jego bezpośrednim przodkiem,
2. jeśli ktoś jest przodkiem bezpośredniego przodka pewnej osoby, to jest także przodkiem tej osoby,
3. każdy jest spokrewniony z każdym swoim przodkiem,
4. każdy jest spokrewniony ze swoim bratem i siostrą,
5. każdy jest spokrewniony z braćmi i siostrami wszystkich osób spokrewnionych ze sobą.

Zad. 3.5

Dokonać skolemizacji następującej formuły:

$$\forall x \forall y\, (P(x,y) \Rightarrow \exists z(\forall y\, Q(y,z) \wedge \neg R(x,z))) \vee \forall z\, Q(x,y,z)$$

Zad. 3.6

Dokonać unifikacji następujących par formuł:

1)
$$P(A, f(g(x))) \wedge Q(g(y), B) \Rightarrow R(x, C)$$
$$P(y, f(v)) \wedge Q(z, B) \Rightarrow R(g(z), z)$$

2)
$$\neg P(z, A, f(y)) \wedge (Q(y, B) \Rightarrow R(C, g(z))) \vee S(f(A), g(B), z)$$
$$\neg P(A, v, f(A)) \wedge (Q(z, x) \Rightarrow R(w, g(A))) \vee S(f(z), g(x), y)$$

4. Wnioskowanie w logice predykatów

4.1. Twierdzenia o dedukcji

W logice pierwszego rzędu również zachodzą oba znane nam już **twierdzenia o dedukcji**:

1. $KB \models \alpha$ wtw. gdy $(KB \Rightarrow \alpha)$ jest tautologią.
2. $KB \models \alpha$ wtw. gdy formuła $(KB \wedge \neg\alpha)$ jest niespełnialna.

Ze względu na potencjalnie nieprzeliczalną liczbę wartościowań dla badanej teorii, która jest możliwa w języku logiki pierwszego rzędu, pojawia się problem z generalną rozstrzygalnością procesu wnioskowania. W logice predykatów zachodzi następujące twierdzenie:

„Problem stwierdzenia, czy dana formuła jest tautologią czy nie, jest problemem nierozstrzygalnym".

Można jedynie pokazać, że:

„istnieje algorytm, który dla zadanej formuły *A* stwierdza, że *A* jest tautologią, pod warunkiem, że tak istotnie jest."

4.2. Postaci normalne formuł i reguły wnioskowania

Procedury wnioskowania zakładają, że formuły występują we właściwej postaci **normalnej**. Pozwala to zdefiniować obliczeniowo efektywną procedurę wnioskowania.

4.2.1. Klauzula Horna i uogólniona reguła odrywania

Dla procedur stosujących regułę *Modus Ponens* (zwaną także *regułą odrywania*) zdania (lub formuły) przyjmują postać tzw. **klauzul Horna**. Klauzula Horna to pojedynczy prosty literał lub implikacja o postaci:

(*koniunkcja prostych literałów*) $\Rightarrow$ prosty literał .

„Prosty" oznacza „pozytywny", nie zanegowany.

Uogólniona reguła odrywania (*General Modus Ponens*) stosowana jest w procedurach wnioskowania dla logiki predykatów. Jest to następująca reguła:

$$\frac{p_1', \cdots, p_n', \qquad p_1 \wedge \cdots \wedge p_n \Rightarrow q}{SUBST(\theta, q)}$$

gdzie $(p_1', \ldots, p_n', p_1, \ldots, p_n, q)$ są literałami, a θ jest podstawieniem takim, że dla każdego $i = 1,2,\ldots,n$:

$\mathrm{SUBST}(\theta, p_i') = \mathrm{SUBST}(\theta, p_i)$.

Np. $p_1' = Król(Jan)$, $p_1 = Król(x)$, $p_2' = Chciwy(y)$, $p_2 = Chciwy(x)$, $q = Zło(x)$

$\theta = \{x/Jan, y/Jan\}$, $\mathrm{SUBST}(\theta, q) = Zło(Jan)$

Zastosowanie uogólnionej reguły odrywania poprzedzone jest zamianą każdej formuły dodawanej do bazy wiedzy do postaci klauzul Horna, wykonywaną w następujących krokach:

1. Przemianowanie zmiennych w formule - każda zmienna w formule związana kwantyfikatorem musi być unikalnie nazwana.
2. Zamiana każdej formuły na postać iloczynu klauzul;

3. Przekształcanie iloczynu klauzul do postaci zbioru klauzul przez eliminację koniunkcji
4. Eliminacja kwantyfikatorów egzystencjalnych
5. Opuszczenie kwantyfikatorów uniwersalnych
6. Przekształcenie do postaci zbioru klauzul zgodnie z zasadami rachunku zdań

Eliminacja uniwersalnego kwantyfikatora

Każde wartościowanie formuły związanej uniwersalnym kwantyfikatorem wynika z tej formuły. Możemy to zapisać w postaci reguły:

$$\frac{\forall_v \alpha}{SUBST(\{v/g\},\alpha)}$$

dla każdej zmiennej v i termu g.

Tym samym kwantyfikator uniwersalny nie nakłada ograniczeń na wartościowanie związanej nim zmiennej. Jeśli nie budzi to wątpliwości, po uprzednim unikalnym przemianowaniu zmiennych i po eliminacji ewentualnych kwantyfikatorów egzystencjalnych, opuszczamy kwantyfikatory uniwersalne.

Eliminacja egzystencjalnego kwantyfikatora

Eliminacja egzystencjalnego kwantyfikatora nosi nazwę **skolemizacji formuły**. Rozróżnimy tu dwa przypadki skolemizacji, zależnie od tego czy w formule występuje kwantyfikator uniwersalny poprzedzający kwantyfikator egzystencjalny, czy też nie.

Eliminacja kwantyfikatora egzystencjalnego nie poprzedzonego żadnym kwantyfikatorem uniwersalnym polega na zastosowaniu następującej reguły wnioskowania: „dla każdej formuły α, zmiennej v i pewnego symbolu stałej K, który nie występuje nigdzie indziej w bazie wiedzy, zachodzi”:

$$\frac{\exists_v \alpha}{SUBST(\{v/K\},\alpha)}$$

Eliminacja kwantyfikatora wiąże się z jednoczesnym przemianowaniem zmiennej v w nim związanej na pewną unikalną stałą K.

Np. z formuły, $\exists x\ Korona(x) \wedge NaGłowie(x, Jan)$, wynika:

$$Korona(C_1) \wedge NaGłowie(C_1, Jan),$$

pod warunkiem, że C_1 jest nowym symbolem stałej zwanej **stałą Skolema**.

Jeśli kwantyfikator egzystencjalny formuły poprzedzony jest kwantyfikatorem uniwersalnym dla pewnej zmiennej x to w miejsce zmiennej za v podstawiamy unikalny symbol funkcji zwanej **funkcją Skolema** o parametrze x. Wyrażamy to w postaci reguły wnioskowania jako:

$$\frac{\forall_x \exists_v \alpha}{SUBST(\{v/F(x)\},\alpha)}$$

4.2.2. Postać normalna CNF i reguła rezolucji

Drugą postacią normalną dla zdań (lub formuł) jest **koniunkcyjna postać normalna** (ang. *Conjunctive Normal Form*, **CNF**) będąca koniunkcją alternatyw literałów.

Uogólniona reguła rezolucja, właściwa dla logiki predykatów, jest postaci:

$$\frac{l_1 \vee ... \vee l_k\ ,\ m_1 \vee ... \vee m_n}{SUBST(\theta,(l_1 \vee ... \vee l_{i-1} \vee l_{i+1} \vee ... \vee l_k \vee m_1 \vee ... \vee m_{j-1} \vee m_{j+1} \vee ... \vee m_n))}$$

gdzie $\text{UNIFY}(l_i, \neg m_j) = \theta$.

Zakłada się, że formuły są w postaci klauzul Horna, tzn. wszystkie l_i, m_i są literałami, a dwie formuły w poprzedniku zostały standaryzowane rozłącznie, tzn. nie mają wspólnych zmiennych.

Np. Z pary formuł w postaci CNF, $\neg Bogaty(x) \vee Nieszczęśliwy(x)$, i , $Bogaty(Jan)$, o unifikowalnych komplementarnie literałach $\neg Bogaty(x)$ i $Bogaty(Jan)$, wnioskujemy zachodzenie formuły: $Nieszczęśliwy(Jan)$, przy zastosowaniu podstawienia: $\theta = \{x/Jan\}$.

4.3. Procedury wnioskowania

W rachunku zdań procedura wnioskowania sprawdza jedynie, czy podane zdanie zapytania wynika z bazy wiedzy czy też nie. Np. ASK(KB, *Brat*(Jan,Piotr)) , Ask(KB, *Lubi*(Anna,Jan)).

W logice predykatów możemy spytać bazę wiedzy o prawdziwość formuły dla wielu obiektów na raz. Mając formułę S pytamy się, czy istnieje podstawienie θ dla którego S wynika z bazy wiedzy. Wywołanie funkcji ASK(KB, S) zwraca wszystkie podstawienia θ takie, że: $KB \models SUBST(\theta, S)$.

Pytania kierowane do bazy wiedzy są typu: kto, gdzie, kiedy?

Np. „*Kto jest bratem Piotra*?": ASK(KB, $\exists$x Brat(x, Piotr)) . Oczekiwana odpowiedź to zbiór podstawień, np.: {{x/Jan}, {x/Stefan}} .

4.3.1. Wnioskowanie poprzez rezolucję

Wnioskowanie stosujące z regułą rezolucji prowadzone jest nie wprost i polega na sprawdzeniu warunków niespełnialności formuły będącej zaprzeczeniem formuły zapytania. Jeśli α jest formułą zapytania to jej zaprzeczenie dodawane jest do bazy wiedzy (jest ona wtedy postaci CNF(KB $\wedge$ $\neg$α)) a następnie stosuje się iteracyjnie regułę rezolucji. Jeśli zostanie wygenerowana formuła pusta to będzie oznaczać, że formuła zapytania zachodzi.

Przykład. Konwersja formuły do postaci CNF. Zdanie „*każdy kto kocha wszystkie zwierzęta jest kochany przez kogoś*" wyrazimy w postaci formuły logiki predykatów jako:

$$\forall x\, [\forall y\, Zwierz(y) \Rightarrow Kocha(x,y)] \Rightarrow [\exists y\, Kocha(y,x)].$$

1. Eliminacja obu implikacji:

$$\forall x\, [\neg \forall y\, \neg Zwierz(y) \vee Kocha(x,y)] \vee [\exists y\, Kocha(y,x)]$$

2. Przesuwamy $\neg$ w prawo: $\neg \forall x\, p \equiv \exists x\, \neg p$, $\neg\, \exists x\, p \equiv \forall x\, \neg p$

$$\forall x\, [\exists y\, \neg(\neg Zwierz(y) \vee Kocha(x,y))] \vee [\exists y\, Kocha(y,x)]$$

$$\forall x\, [\exists y\, \neg\neg Zwierz(y) \wedge \neg Kocha(x,y)] \vee [\exists y\, Kocha(y,x)]$$

$$\forall x\, [\exists y\, Zwierz(y) \wedge \neg Kocha(x,y)] \vee [\exists y\, Kocha(y,x)]$$

3. Standaryzacja rozłączna zmiennych: każdy kwantyfikator korzysta z innej zmiennej

$$\forall x\, [\exists y\, Zwierz(y) \wedge \neg Kocha(x,y)] \vee [\exists z\, Kocha(z,x)]$$

4. Skolemizacja: każda egzystencjalna zmienna (i jej kwantyfikator) jest zastępowana przez funkcję Skolema dla zmiennej należącej do poprzedzającego kwantyfikatora uniwersalnego:

$$\forall x\, [Zwierz(F(x)) \wedge \neg Kocha(x,F(x))] \vee Kocha(G(x),x)$$

5. Pomijamy uniwersalny kwantyfikator:

$$[Zwierz(F(x)) \wedge \neg Kocha(x,F(x))] \vee Kocha(G(x),x)$$

6. Rozdzielamy $\vee$ względem $\wedge$:

$$[Zwierz(F(x)) \vee Kocha(G(x), x)] \wedge [\neg Kocha(x,F(x)) \vee Kocha(G(x),x)]$$

Uzyskaliśmy postać CNF, w tym przypadku jest to iloczyn dwóch klauzul.

7. Ewentualnie z klauzul usuwamy każdy literał o postaci $\neg\mathrm{True}$ i False. Usuwamy też klauzule zawierające literał $\neg\mathrm{False}$ lub True.

4.3.2. Wnioskowanie w przód

Wnioskowanie w przód zwykle stosujemy wtedy, gdy nowa formuła p zostaje dodana do bazy wiedzy *KB* (wtedy implementujemy to wnioskowanie jako element funkcji **TELL**). Poniższa funkcja wnioskowania WPRZOD() (tabela 4-1) korzysta z reguły „*uogólnione modus ponens*" i znajduje ona nowe literały, które wynikają z $KB \cup \{ p \}$.

Tab. 4-1. Funkcja wnioskowania w przód w logice predykatów.

```
function WPRZOD(KB, p) zwraca Tak/Nie
{
        if (w KB istnieje już wariant formuły p) then return Tak;
        Dodaj p do KB;
        for ((p1 ∧ p2 ∧ … ∧ pn ⇒q) ∈ KB takie, że zachodzi ∃i UNIFY(pi, p) = θ ) do
                WNIOSKUJ(KB, [p1,..., pi-1,pi+1,…,pn ], q, θ );
        return Nie; // nie było jeszcze zadanej formuły w bazie wiedzy
}
```

```
procedure WNIOSKUJ(KB, warunki, konkluzja, θ)
{
        if (warunki == []) then
        WPRZOD(KB, SUBST(θ, konkluzja));
        else
        for (każde p' ∈ KB takie, że  UNIFY(p', SUBST(θ , Pierwszy(warunki)))) = θ2  do
                        WNIOSKUJ(KB, Reszta(warunki), konkluzja, COMPOSE(θ, θ2));
}
```

4.3.3. Wnioskowanie wstecz

Wnioskowanie wstecz stosujemy wtedy, gdy chcemy otrzymać odpowiedź na pytanie zadane do bazy wiedzy (implementujemy funkcję **ASK**). W tab. 4-2 podano schemat procedury wnioskowania wstecz, w którym wywoływana jest rekurencyjna podfunkcja. Parametry funkcji WSTECZ_LISTA() to: baza wiedzy *KB*, lista celów [q], szukane podstawienie {}. Jest to funkcja rekurencyjna, wywoływana najpierw dla celu (formuły zapytania) a następnie dla kolejnych podcelów, przy jednoczesnym rozszerzaniu zbioru podstawień.

Tab. 4-2. Funkcja wnioskowania wstecz w logice predykatów.

```
function WSTECZ(KB, q) : zwraca zbór podstawień;
{
        WSTECZ_LISTA(KB, [q], {});
}
```

Tab. 4-2 (c.d.)

```
function WSTECZ_LISTA(KB, cele, θ) : zwraca zbiór podstawień wynik (i odpowiednio rozszerza
                                                    zbiór podstawień θ);
{
        wynik ← ∅;
        if (lista cele jest pusta) then return {θ};
        q←SUBST(θ, Perwszy(cele));
        for (każdy r ∈ KB taki, że STAND_ROZŁĄCZNA(r) = (p1 ∧ p2 ∧ … ∧ pn ⇒q') ∈ KB
                i gdy zachodzi θ' ←UNIFY(q, q') ) do
        wynik ← WSTECZ_LISTA(KB, SUBST(θ', [p1,...,pn|Reszta(cele)], COMPOSE(θ, θ'))
                ∪ wynik;
        return wynik;
}
```

Wnioskowanie wstecz jest potencjalnie niepełne z powodu możliwości istnienia pętli zależności w grafie I-LUB. Dla zabezpieczenia przed taką sytuacją należy sprawdzać aktualny cel z każdym celem pamiętanym na stosie i unikać wielokrotnego umieszczania celu na stosie. Procedura jest tez potencjalnie nieefektywna na skutek możliwego powtarzania celów już opracowanych (zarówno z pozytywnym jak i negatywnym wynikiem). Zabezpieczeniem przed tym jest pamiętanie w procedurze poprzednio wywoływanych celów (co wiąże się z potrzebą dodatkowej pamięci).

4.4. System logicznego wnioskowania

4.4.1. Przykłady

Przykłady logicznych systemów wnioskowania

- Dowodzenie twierdzeń (ang. *theorem provers*) (np. AURA, OTTER) – stosują rezolucję w logice predykatów dla dowodzenia twierdzeń w zadaniach matematycznych oraz do realizacji systemu zapytań do bazy wiedzy.
- Języki programowania w logice (np. PROLOG) – zwykle stosują wnioskowanie wstecz, nakładają ograniczenia na język logiki i dysponują dodatkowymi proceduralnymi elementami (np. funkcje wejścia/wyjścia).

4.4.2. Zadania systemu

System z bazą wiedzy wyrażoną w języku logiki stanowi implementację systemu logicznego wnioskowania, czyli obok języka do deklaratywnej reprezentacji wiedzy posiada także odpowiedni mechanizm wnioskowania. Zdefiniujmy podstawowe **zadania** realizowane przez taki system:

- WPROWADŹ – operacja wprowadzania nowego faktu do bazy wiedzy – realizacja w postaci funkcji TELL;
- WNIOSKUJ nowe fakty z połączenia dotychczasowej zawartości bazy wiedzy i nowo wprowadzonych faktów – realizacja w systemie opartym o wnioskowanie wprzód jako część funkcji TELL;
- DECYDUJ czy zapytanie WYNIKA z bazy wiedzy – realizacja w postaci funkcji ASK;
- DECYDUJ czy zapytanie ISTNIEJE w bazie wiedzy – ograniczona wersja funkcji ASK;
- USUŃ – usuwa zdanie z bazy wiedzy z różnych przyczyn – zdanie jest nieprawdziwe, już niepotrzebne lub modyfikacja wynika ze zmiany w środowisku.

4.4.3. Implementacja

O efektywności procedur wnioskowania decyduje głównie właściwa implementacja bazy wiedzy i tych części funkcji TELL i ASK, które bezpośrednio odwołują się do KB.

Implementacja zdań i formuł w KB

Wprowadzamy bazowy typ danych COMPOUND, który reprezentuje powiązanie operatora (czyli predykatu, funkcji lub spójnika logicznego) z listą argumentów (czyli z termami lub zdaniami). Posiada on pola dla reprezentacji operatora (OP) i argumentów (ARGS).

Np. niech istnieje zdanie „c": $P(x) \wedge Q(x)$.

Wtedy: $OP[c] = \wedge$ i $ARGS[c] = [P(x), Q(x)]$.

Funkcje dostępu do KB

Wprowadzamy funkcje dostępu do bazy danych: $STORE(KB, S)$ i $FETCH(KB, Q)$.

Przy nieuporządkowanej bazie wiedzy złożoność obliczeniowa obu funkcji wynosi O(*n*), gdzie *n* jest liczbą elementów w bazie wiedzy. Dla osiągnięcia efektywnego wykonania tych często wołanych funkcji należy uporządkować elementy stosując, np. indeksowanie z wykazem asocjacyjnym lub indeksowanie w postaci drzewa.

Algorytm UNIFIKACJI

4.5. Realizacja celu jako rozwiązanie problemu

4.5.1. Przykład

Działanie agenta w „świecie Wumpusa" jest celowe – agent ma osiągnąć dwa kolejne **cele**.

- Cel 1: Wybierając najlepszą (najbezpieczniejszą lub najtańszą) akcję w danej sytuacji, agent potrafi rozsądnie poruszać się po jaskini w **celu dotarcia do złota**. Nie ma tu jawnej **lokalizacji miejsca** lecz jedynie warunek wykonania najlepszej akcji – „pobierz złoto jeśli to możliwe". Agent poznaje środowisko i bezpiecznie przemieszcza się dopóki nie znajdzie złota.
- Cel 2: Po „pobraniu złota" kolejnym **celem** agenta staje się bezpieczny powrót do kwadratu startowego i wydostanie się z jaskini. Cel zostaje jawnie określony w postaci własności dodanej do bazy wiedzy w odpowiedniej sytuacji:

$$\forall_{\mathbf{x},s}\ TrzymaZłoto(\mathbf{x}, s) \Rightarrow LokalizacjaCelu([1,1], s)$$

4.5.2. Techniki realizacji celu

Rozwiązanie problemu (systemu ekspertowego lub agentowego) postrzegamy abstrakcyjnie jako sekwencję akcji realizującą zadany cel. W systemie logicznego wnioskowania mamy do dyspozycji trzy główne techniki rozwiązania zadania realizacji celu:

- **Wnioskowanie**

  Jest to ogólna technika, ale najmniej efektywna z trzech. Sterowanie systemu z bazą wiedzy (realizacja funkcji agenta) jest bardzo proste – polega na ciągłym zadawaniu do bazy wiedzy zapytania o najlepszą akcję w danej sytuacji. Oczywiście w każdej nowej sytuacji do bazy wiedzy dodawane są nowe obserwacje i wykonane akcje.

- **Przeszukiwanie** przestrzeni stanów problemu

  Podsystem sterowania pełni znacznie większą rolę niż poprzednio. Problem przedstawiony zostaje w postaci przestrzeni stanów. Ujawniana jest wiedza o powiązaniach pomiędzy stanami środowiska i wykonywanymi akcjami. Wybór akcji jest efektem realizacji przez

podsystem sterowania odpowiedniej strategii przeszukiwania przestrzeni stanów (każdemu przejściu pomiędzy stanami odpowiada wykonanie akcji).

- **Planowanie działań**

  Jest to dodatkowy mechanizm wspomagający podsystem sterowania w efektywnym wyborze akcji wtedy, gdy przestrzeń stanów problemu jest bardzo duża. Polega on na wyznaczeniu sekwencji akcji (najczęściej wiodącej do kolejnego podcelu) w oparciu o dodatkową informację o problemie. Np. ujawniane są własności poszczególnych stanów („otworzenie stanu") i określany jest sposób oddziaływania każdej akcji na te własności, co pozwala skupić się na akcjach relewantnych do danej sytuacji. Planowanie jest zwykle rozwiązywane poprzez przeszukiwanie przestrzeni planów, właściwych dla problemu.

W kolejnym punkcie pokażemy, jak może wyglądać ogólna funkcja sterująca w systemie z bazą wiedzy, wyrażoną w języku logiki, posługująca się jedynie mechanizmem wnioskowania dla wyboru akcji. Podstawowe strategie przeszukiwania przestrzeni stanów są omawiane w części II tego podręcznika. Natomiast zagadnienie planowania działań, jako zaawansowana technika Sztucznej Inteligencji, nie będzie tu omawiane (jest omawiane na studiach magisterskich).

4.5.3. Wnioskowanie jako realizacja celu

Ogólna funkcja sterująca w systemie z bazą wiedzy wyrażoną w języku logiki, posługująca się jedynie wnioskowaniem dla wyboru akcji, została przedstawiona w tabeli 4-3. Funkcja agenta posiada podfunkcję „LISTAMOŻLIWYCHAKCJI()". Realizacja tej funkcji jest trywialna – zwraca ona zawsze pełną listę możliwych akcji. Funkcja ASK uruchamia procedurę wnioskowania właściwą dla języka bazy wiedzy i sprawdzającą zachodzenie formuły zapytania utworzonej w funkcji „UTWÓRZZAPYTANIEAKCJI()". Procedura wnioskowania bezpośrednio nie wybiera akcji. Pośredni wybór akcji osiągniemy jedynie wtedy, gdy w bazie wiedzy właściwie zdefiniujemy aksjomaty wyboru akcji.

Tab. 4-3. Program sterujący w systemie z bazą wiedzy z wyborem akcji bezpośrednio w wyniku wnioskowania

```
funkcja KBSYSTEMZWYBOREMAKCJI(obserwacja)
zwraca: akcja
{ static: KB, // baza wiedzy
          t; // licznik czasu, początkowo wynosi 1
  TELL(KB, UTWÓRZZDANIEOBSERWACJI(obserwacja, t));
  for each (akcja in LISTAMOŻLIWYCHAKCJI(KB, t)) {
          if (ASK(KB, UTWÓRZZAPYTANIEAKCJI(t, akcja) ) {
                    t ← t+1;
                    TELL(KB, UTWÓRZZDANIEAKCJI(akcja, t));
                    return akcja;
          }
}
```

4.6. Pytania

1. Na czym polega **uogólniona reguła odrywania** („modus ponens")?
2. Omówić procedurę wnioskowania „w przód" w logice predykatów.
3. Omówić procedurę wnioskowania „wstecz" w logice predykatów.
4. Na czym polega **reguła rezolucji** w logice predykatów?

5. Omówić elementy implementacji logicznego systemu wnioskowania.
6. Omówić techniki realizacji celu przez system Sztucznej Inteligencji.

4.7. Zadania

Zad. 4.1

Przekształcić poniższą formułę do postaci normalnej dla wnioskowania **metodą rezolucji**:

$\forall x \quad B(x) \Leftrightarrow (\exists y\ P(y,x) \vee L(x,y))$

Podać **krok-po-kroku** realizowane przekształcenia. **Wyjaśnić** realizowane przekształcenia.

Zad. 4.2

Zakładamy, że agent w *świecie Wumpusa* znajduje się w kratce [1,1] w chwili t=1. Niech agent czuje smród i wiatr (i brak innych).

A) Jaki jest efekt wnioskowania w rachunku zdań o nowych faktach, w wyniku przekazania powyższej obserwacji do bazy wiedzy, tzn. jak **powiększy się zbiór zdań** w bazie wiedzy i **w wyniku jakiego wnioskowania**?

B) Zilustrować odpowiedź krokami wnioskowania.

Zad. 4.3

Zakładamy regułę modelowanego świata:

- *Według prawa przestępstwo popełnia Amerykanin, który sprzedaje broń wrogim narodom*.

Znamy fakty:

- *Kraj Nono, wróg Ameryki, posiada rakiety, które sprzedał im pułkownik West, będący Amerykaninem.*

Sprawdzić, czy *„pułkownik West jest przestępcą”,* stosując procedurę wnioskowania w przód w logice predykatów.

Zad. 4.4

Zilustrować działanie funkcji **wnioskowania wstecz** w logice predykatów na przykładzie zbioru formuł z zadania 4.3 i zapytania „ ∃x *Przestępca*(x)”?

Zad. 4.5

Zilustrować działanie funkcji **wnioskowania metodą rezolucji** w logice predykatów na przykładzie zbioru formuł z zad. 4.3 i zapytania „*Przestępca*(West)”?

Zad. 4.6

Sprawdzić, czy z bazy wiedzy KB wynikają zapytania q_1 i q_2.

$$P(x,y) \wedge Q(y,z) \Rightarrow R(x,y)$$

KB reguły: $R(x_1,y_1) \wedge S(z_1,v) \wedge W(y_1,v) \Rightarrow W(x_1,z_1)$

$$\neg Q(x_2,y_2) \Rightarrow Q(y_2,x_2)$$

KB obserwacje: $P(Ania,Beata), \quad \neg Q(Czarek,Beata)$

$S(Darek,Ewa), \quad W(Czarek,Ewa)$

Zapytania:

q_1: $W(a,d)$

q_2: $W(d,e) \Rightarrow W(a,d)$

5. System ekspertowy

5.1. Inżynieria wiedzy

Inżynieria wiedzy zajmuje się metodologią konstruowania i korzystania z systemów z bazą wiedzy. Jednym z jej przejawów jest „programowanie w logice", uznawane też za jeden z 4 głównych paradygmatów programowania. Poniższa tabelka 5-1 pokazuje zasadnicze różnice pomiędzy inżynierią oprogramowania a inżynierią wiedzy.

Tab. 5-1. Porównanie inżynierii oprogramowania i inżynierii wiedzy.

Inżynieria oprogramowania	Inżynieria wiedzy
Języki programowania	Języki reprezentacji wiedzy
Dane	Bazy wiedzy
Operacje	Procedury wnioskowania (dowodzenia zdań)
Wykonanie programu	Wnioskowanie (dowodzenie zdania zapytania)

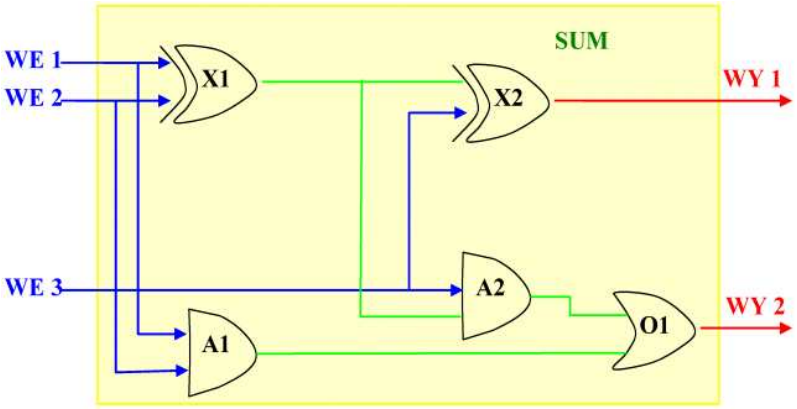
5.1.1. Zasady wyznaczania bazy wiedzy

Zgodnie z zasadami inżynierii w procesie tworzenia bazy wiedzy, wyrażonej w języku logiki, wyróżnimy następujące etapy:

1. Identyfikacja zadania.
2. Zebranie odpowiedniej wiedzy o dziedzinie (jakie obiekty i fakty modelowanej dziedziny należy reprezentować w systemie a jakie są zbędne).
3. Wybór słownika dla relacji, funkcji i stałych (ontologia).
4. Zakodować ogólną wiedzę o dziedzinie (aksjomaty).
5. Zakodować opis specyficznego zadania (formuły atomowe).
6. Testowanie - wysyłać zapytania do procedury wnioskowania i odbierać jej odpowiedzi.
7. Przeanalizować wyniki testowania i ewentualnie poprawiać bazę wiedzy .

5.1.2. Przykład

Definiowanie bazy wiedzy dla układu elektronicznego - sumatora jednobitowego (rys. 5.1):



Rys. 5.1: Schemat sumatora jednobitowego

1)Identyfikacja zadania:

sprawdzenie, czy układ wykonuje prawidłową operację dodawania?

2) Zebranie odpowiedniej wiedzy o zasadzie działania:

- zauważenie tego, że układ składa się z połączeń i bramek logicznych;
- określenie typów stosowanych bramek (AND, OR, XOR, NOT)
- bez znaczenia są takie cechy, jak: rozmiar, kształt, kolor, koszt bramek

3) Wybór słownika:

- stałe oznaczają bramki; np. X1, X2.
- typ bramki podany będzie dzięki funkcji; np. Typ(X1) = XOR;
  inne alternatywne (ale gorsze) rozwiązanie – poprzez predykaty: Typ(X1, XOR), XOR(X1).
- wejścia / wyjścia bramki podają kolejne funkcje; np. We(1,X1), Wy(1,X1).
- istnienie połączenia bramek wyznacza relacja: Połączenie(Wy(..), We(..)).
- wartości sygnału to obiekty 1, 0 – podaje je funkcja: Sygnał(Wy(..)).

4) Zakodowanie ogólnej wiedzy o dziedzinie (zasady rządzące dziedziną zadane są w postaci formuł korzystających z symboli przyjętego słownika):

$\forall t1,t2\ \text{Połączenie}(t1, t2) \Rightarrow \text{Sygnał}(t1) = \text{Sygnał}(t2)$

$\forall t\ \text{Sygnał}(t) = 1 \vee \text{Sygnał}(t) = 0$

$1 \neq 0$

$\forall t1,t2\ \text{Połączenie}(t1, t2) \Rightarrow \text{Połączenie}(t2, t1)$

$\forall g\ \text{Typ}(g) = \text{OR} \Rightarrow$

$(\ \text{Sygnał}(\text{Wy}(1,g)) = 1 \Leftrightarrow \exists n\ \text{Sygnał}(\text{We}(n,g)) = 1\ )$

$\forall g\ \text{Typ}(g) = \text{AND} \Rightarrow$

$(\text{Sygnał}(\text{Wy}(1,g)) = 0 \Leftrightarrow \exists n\ \text{Sygnał}(\text{We}(n,g)) = 0\ )$

$\forall g\ \text{Typ}(g) = \text{XOR} \Rightarrow$

$(\text{Sygnał}(\text{Wy}(1,g)) = 1 \Leftrightarrow \text{Sygnał}(\text{We}(1,g)) \neq \text{Sygnał}(\text{We}(2,g))\ )$

$\forall g\ \text{Typ}(g) = \text{NOT} \Rightarrow (\text{Sygnał}(\text{Wy}(1,g)) \neq \text{Sygnał}(\text{We}(1,g))\ )$

5) Zakodowanie specyficznego problemu, czyli określenie własności konkretnie badanego układu elektronicznego (logiczne fakty):

$Typ(X_1) = XOR$ $Typ(X_2) = XOR$

$Typ(A_1) = AND$ $Typ(A_2) = AND$

$Typ(O_1) = OR$

$Połączenie(Wy(1,X_1),We(1,X_2))$, $Połączenie\ (We(1,C_1),We(1,X_1))$

$Połączenie(Wy(1,X_1),We(2,A_2))$, $Połączenie\ (We(1,C_1),We(1,A_1))$

$Połączenie(Wy(1,A_2),We(1,O_1))$, $Połączenie\ (We(2,C_1),We(2,X_1))$

$Połączenie(Wy(1,A_1),We(2,O_1))$, $Połączenie\ (We(2,C_1),We(2,A_1))$

$Połączenie(Wy(1,X_2), Wy(1,C_1))$, $Połączenie\ (We(3,C_1),We(2,X_2))$

$Połączenie(Wy(1,O_1), Wy(2,C_1))$, $Połączenie\ (We(3,C_1),We(1,A_2))$

6) Testowanie – generowanie zapytań kierowanych do procedury wnioskowania.

Np. pytanie jest postaci: „jakie sygnały wejściowe spowodują, że wyjście pierwsze bramki C1 będzie „0" a jej wyjście drugie „1" ?":

$\exists i1,i2,i3$ Sygnał(We(1,C1))=i1 $\wedge$ Sygnał(We(2,C1))=i2 $\wedge$ Sygnał(We(3,C1))=i3 $\wedge$ Sygnał(Wy(1,C1))=0 $\wedge$ Sygnał(Wy(2,C1))=1

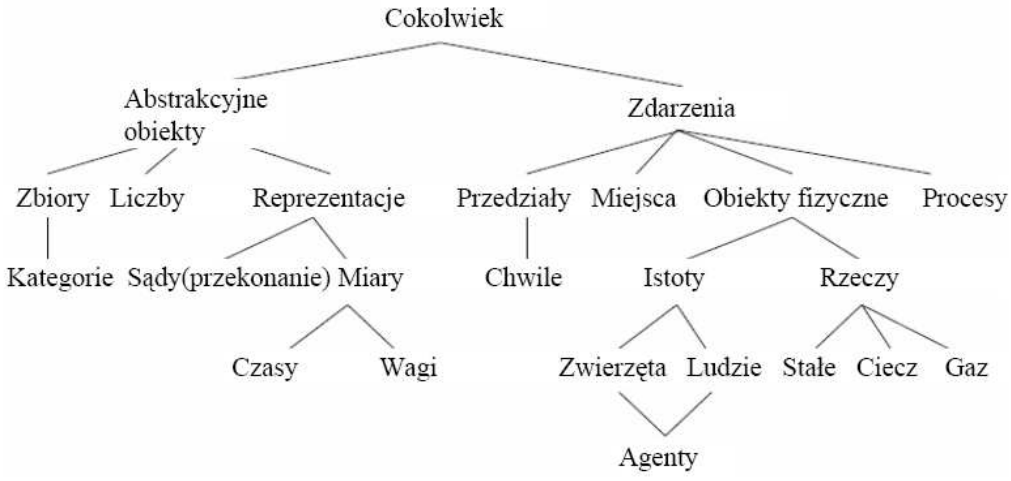
Poprawna odpowiedź to zbiór podstawień:

$\theta = \{ \{i1/1, i2/1, i3/0\}, \{i1/1, i2/0, i3/1\}, \{i1/0, i2/1, i3/1\} \}$

7) Ewentualne poprawianie bazy wiedzy. Można też opuścić oczywiste związki, jak np. $1 \neq 0$.

5.2. Ontologia języka

Ontologia to formalizacja (modelowanie) fundamentalnych pojęć wiedzy (obiektów, relacji), tzn. takich, które występują w wielu dziedzinach zastosowań. Np. są to pojęcia: kategorie, miary, czas itp. (rys. 5.2).



Rys. 5.2: Przykład kategoryzacji pojęć

Kategoria

Kategoria - zbiór obiektów o podobnych cechach.

Jak reprezentować kategorie?

1) Przy użyciu 1-argumentowych predykatów. Np. *Pomidor*(*a*).

2) Stosując technikę reifikacji – predykat lub funkcja traktowane są jak obiekt (stała) języka. Np. *Pomidory* - stała interpretowana jako obiekt reprezentujący zbiór wszystkich pomidorów.

Kategorie w logice predykatów

$Pomidor_{12} \in Pomidory$ (obiekt jest elementem kategorii)

$\forall x \; x \in Pomidory \Rightarrow x \in Owoce$ (relacja dziedziczenia pomiędzy kategoriami *Pomidory* i *Owoce*)

$\forall x \; x \in Pomidory \Rightarrow Czerwone(x) \wedge Okrągłe(x)$ (elementy kategorii posiadają własności)

Miary

Miary – abstrakcyjne obiekty dla wyrażenia własności obiektów fizycznych.

Miary ilościowe przedstawiamy jako połączenie funkcji wyrażającej jednostkę miary z liczbą. Np. $Długość(L_1) = Cale(1.5)$

Konwersje pomiędzy jednostkami, np.: $\forall l \; Centymetry(2.54*l) = Cale(l)$

$\forall t \; Celsiusz(t) = Fahrenheit(32+1.8*t)$

Charakterystyka obiektu za pomocą miary, np.: $Waga(Pomidor_{12}) = Kilogramy(0.16)$

$\forall d \; d \in Dni \Rightarrow (CzasTrwania(d) = Godziny(24))$

Należy rozróżniać pomiędzy jednostkami miary a narzędziami operacji na danej wielkości. Np.

$\forall b \; b \in RachunekWDolarach \Rightarrow (JednostkaRachunku(b) = \$(1.00))$

Miary jakościowe są trudniejsze do użycia niż ilościowe, gdyż nie posiadają ogólnie przyjętej skali liczbowych wartości. Można je jednak porządkować. Np.:

$\forall e_1, e_2 \; e_1 \in Ćwiczenia \wedge e_2 \in Ćwiczenia \wedge Autor(e_1, Norvig) \wedge Autor(e_2, Russel)$

$\Rightarrow Trudność(e_1) > Trudność(e_2)$.

Obiekty złożone

Relacja „części-do-całości”. Np.

$CzęśćCałość(Budapeszt, Węgry)$,

$CzęśćCałość(Węgry, EuropaŚrodkowa)$,

$CzęśćCałość(EuropaŚrodkowa, Europa)$

$\forall x,y,z \; CzęśćCałość(x,y) \wedge CzęśćCałość(y, z) \Rightarrow CzęśćCałość(x,z)$

Obiekt złożony – to obiekt, który posiada części. Możemy określić strukturę obiektu, przez podanie jego części i relacji pomiędzy częściami.

Związek obiektów – to obiekt, posiadający części ale bez struktury. Np. obiekt złożony z 3 jabłek:

$Związek(\{Jabłko_1, Jabłko_2, Jabłko_3\})$

Zdarzenia (*events*)

Do reprezentowania zmian możemy użyć „zdarzeń”. Zdarzenie to element *czasoprzestrzeni* – coś co odbywa się jednocześnie w czasie i przestrzeni – to jakby ciągła wersja rachunku sytuacyjnego.

Przedział (*interval*) – jest to specjalne zdarzenie posiadające jedynie wymiar czasowy – przedział czasu. Zawiera w sobie, jako pod-zdarzenia, wszystkie te zdarzenia, które wystąpiły w tym przedziale czasu. Np.: *PodZdarzenie*(WojnaŚwiatowa2, WiekXX)

Zdarzenia możemy pogrupować w kategorie. Np. skorzystamy z kategorii *Podróże* aby wyrazić zdarzenie odbycia konkretnej podróży:

$$\exists j \; j \in Podróże \wedge Start(Warszawa, j) \wedge Cel(NewYork, j) \wedge$$

$$Podróżny(Kowalski, j) \wedge PodZdarzenie(j, Wczoraj).$$

Często interesują nas nie tyle same nazwy kategorii co ich właściwości. Skorzystamy w tym celu z formuł złożonych. Np. symbol *Go* będzie wyznaczał pewien rodzaj podróży - niech $Go(x,s,c)$ oznacza, że osoba *x* podróżuje z miejsca *s* do miejsca *c*:

$$\forall e,x,s,c \;\; e \in Go(x,s,c) \Leftrightarrow [e \in Podróże \wedge Podróżny(x,e) \wedge Start(s,e) \wedge Cel(c,e)].$$

Użyjmy notacji $E(c,i)$, aby wyrazić to, że zdarzenie kategorii *c* jest pod-zdarzeniem zdarzenia (lub przedziału) *i*:

$$\forall c,i \;\; E(c,i) \Leftrightarrow \exists e \; e \in c \wedge PodZdarzenie(e,i).$$

Np: $E(Go(Kowalski,Warszawa,NewYork),Wczoraj)$

Miejsca

Miejsca, podobnie jak przedział, stanowią obszar w czaso-przestrzeni, ale o niezmiennym położeniu „rozciągniętym w czasie. Relacje pomiędzy miejscami wyrazimy np. w postaci predykatu *W* („miejsce zawarte w"). Np. *W* (Warszawa, Polska)

Wprowadźmy funkcje odwzorowującą obiekty i miejsca. Np. funkcja *Miejsce* odwzorowuje obiekt na najmniejsze miejsce, zawierające ten obiekt:

$$\forall x,l \;\; Miejsce(x) = l \Leftrightarrow Jest(x,l) \wedge \forall l_2 \; Jest(x,l_2) \Rightarrow W(l,l_2).$$

Procesy

Proces to zdarzenie kategorii *c*, którego każde pod-zdarzenie (w mniejszym przedziale czasu) jest zdarzeniem tej samej kategorii *c.* W ten sposób wyrazimy zdarzenie ciągłe w czasie. Np. $Lot(Kowalski)$ wzięty jako całość lub jego fragment, są to zawsze zdarzenia tej samej kategorii *Lot*.

W odniesieniu do zdarzenia ciągłego w czasie (procesu) możemy zastosować tę samą notacją co poprzednio do zdarzeń dyskretnych. Np. lot odbył się wczoraj:

$$E(Lot(Kowalski),Wczoraj).$$

Często jednak chcemy wyrazić, że proces trwał dokładnie w jakimś przedziale czasu. Użyjemy notacji $T(c,i)$ na wyrażenie faktu, że zdarzenie *c* odbyło się dokładnie w przedziale *i*.

Np. $T(Pracuje(Jan), DzisiejszaPoraLunchu)$.

Notacja dla łączenia zdarzeń i czasu

Nie możemy napisać: $T(Jest(A_1,Loc_1) \wedge Jest(A_2,Loc_2), i)$, gdyż pierwszy argument predykatu *T* nie jest termem tylko formułą. Ogólnie zauważymy, że w wyrażeniu postaci $T(p \wedge q, e)$ - $p \wedge q$ jest formułą, a nie termem wskazującym na kategorię zdarzeń. Dlatego powinniśmy wprowadzić symbol funkcji *And* zdefiniowany aksjomatem:

$$\forall p,q,e \;\; T(And(p,q), e) \Leftrightarrow T(p,e) \wedge T(q,e) ,$$

czyli funkcja *And* zwraca kategorię złożonego obiektu. Możemy teraz napisać:

$$T(And(Jest(A_1,Loc_1), Jest(A_2,Loc_2)), i) .$$

Podobnie wprowadzimy, np. symbol funkcji *OR* dla połączenia dwóch zdarzeń (wzajemnie wykluczających się) alternatywą:

$$\forall p,q,e \;\; T(OR(p,q), e) \Leftrightarrow T(p,e) \vee T(q,e) .$$

Czas

Podstawowe pojęcie w tym zakresie to: **przedział** (odcinek) czasu. Z przedziałem związany jest jego czas trwania (długość). Przedział o długości zero nazwiemy **chwilą** czasu.

$$\forall i \;\; i \in \mathit{Przedziały} \Rightarrow [i \in \mathit{Chwile} \Leftrightarrow \mathit{Trwa}(i) = 0] \; .$$

Wprowadźmy pewne funkcje, wyznaczające chwile czasu na globalnej osi czasu:

Start(*i*) – początkowa chwila przedziału,

End(*i*) – końcowa chwila przedziału,

Trwa(*i*) – różnica pomiędzy końcową a początkową chwilą czasu,

Czas(*j*) – wyznacza punkt na osi czasu dla zadanej chwili.

Określmy możliwe położenia wzajemne dwóch odcinków czasu:

$\forall i,j \;\; \mathit{Spotyka}(i,j) \Leftrightarrow \mathit{Czas}(\mathit{End}(i)) = \mathit{Time}(\mathit{Start}(j))$.

$\forall i,j \;\; \mathit{Przed}(i,j) \Leftrightarrow \mathit{Czas}(\mathit{End}(i)) < \mathit{Time}(\mathit{Start}(j))$.

$\forall i,j \;\; \mathit{Po}(j,i) \Leftrightarrow \mathit{Przed}(i,j)$.

$\forall i,j \; \mathit{Podczas}(i,j) \Leftrightarrow \mathit{Czas}(\mathit{Start}(j)) \leq \mathit{Czas}(\mathit{Start}(i)) \wedge \mathit{Czas}(\mathit{End}(i)) \leq \mathit{Czas}(\mathit{End}(j))$

$\forall i,j \;\; \mathit{Przecina}(i,j) \Leftrightarrow \exists k \; \mathit{Podczas}(k,i) \wedge \mathit{Podczas}(k,j)$.

Akcje

Relacje pomiędzy przedziałami czasu wykorzystywane są przy definiowaniu akcji. Zwykle polega to na podaniu przedziału czasu, w którym dane zdarzenie zachodzi.

Przykłady. Kiedy 2 osoby są ze sobą w związku to mogą zajść w przyszłości akcje: *Ślub* lub *RozpadZwiązku*.

$\forall x,y,i_0 \; T(\mathit{Związek}(x,y),i_0) \Rightarrow \exists i_1 \; [\mathit{Spotyka}(i_0,i_1) \vee \mathit{Po}(i_1,i_0)] \wedge T(\mathit{OR}(\mathit{Ślub}(x,y), \mathit{RozpadZwiązku}(x,y)),i_1)$.

Uwaga: OR(*Ślub*(*x,y*), *RozpadZwiązku*(*x,y*)) jest termem wskazującym na kategorię zdarzeń.

Po akcji *Ślub* dwoje ludzi staje się małżonkami:

$\forall x,y,i_0 \;\; T(\mathit{Ślub}(x,y),i_0) \Rightarrow \exists i_1 \; T(\mathit{Małżonek}(x,y),i_1) \wedge \mathit{Spotyka}(i_0,i_1)$

Wynikiem akcji *Go* jest zalezienie się w drugim miejscu:

$\forall x,a,b,i_0 \; \exists i_1 \; T(\mathit{Go}(x,a,b),i_0) \Rightarrow \mathrm{T}(W(x,b),i_1) \wedge \mathit{Spotyka}(i_0,i_1)$

Fluenty

Fluenty są to obiekty, których właściwości zmieniają się w czasie. Np. *Powierzchnia*(*Polska*) – obiekt przyjmujący różne wartości w różnych okresach czasu, *Prezydent*(*USA*) – obiekt wskazujący na różne osoby w różnych momentach czasu.

Fluenty pozwalają na zwarte wyrażenie wspólnych cech. Np. dla wyrażenia faktu, że prezydenci byli mężczyznami w 19 wieku napiszemy: *T*(*Mężczyzna*(*Prezydent*(*USA*))*,19tyWiek).* Np. dla wyrażenia powierzchni Polski w określonym roku napiszemy:

T(*Wartość*(*Powierzchnia*(*Polska*)*, KmKw*(*621000*))*,AD1426*)

T(*Wartość*(*Powierzchnia*(*Polska*)*, KmKw*(*312000*))*,AD1970*)

Sądy (przekonania)

Modelujemy przekonania (*sądy*) agenta za pomocą relacji typu: *Wierzy*(Agent, *x*), *Wie*(Agent, *x*), *Chce*(Agent, *x*). Np. chcemy tym wyrazić, że: „*a* wie, że *p", „a* wierzy, że *p", „a* chce aby, *p"*. Czyli chcemy wyrazić relację pomiędzy agentem i formułą (lub zdaniem).

Ale możemy napisać, *Wierzy*(*Agent, x*), jedynie pod warunkiem, że *x* jest termem. Np. *Wierzy*(*Agent, Lata*(*Superman*)) jest prawidłowe pod warunkiem, że *Lata*(*Superman*) jest termem. Dlatego należy dokonać najpierw *reifikacji* predykatu *Lata*(*Superman*) do postaci *obiektu-fluentu*.

Związek pomiędzy „przekonaniem" a „wiedzą" może być rożnie definiowany. Filozofowie definiują wiedzę jako „udowodniona prawdziwość przekonania". Wyrazimy to następująco:

$$\forall a,p \;\; Wiedza(a,p) \Leftrightarrow Wierzy(a,p) \wedge T(p) \wedge T(KB(a) \Rightarrow p)$$

5.3. System ekspertowy

System ekspertowy można zaimplementować w postaci systemu z bazą wiedzy – dysponuje językiem deklaratywnej reprezentacji wiedzy i mechanizmem wnioskowania. W zależności od sposobu reprezentacji wiedzy wyróżnimy następujące kategorie systemów ekspertowych:

1. Logiczne systemy ekspertowe – przykładami są systemy dowodzenia twierdzeń i systemy stosujące język PROLOG.
2. Systemy regułowe (np. OPS5, CLIPS, SOAR) – stosują formuły implikacji dla reprezentacji warunków wykonania i opisu akcji; wśród akcji są operacje wprowadzania i usuwania do/z bazy wiedzy i operacje we/wy; stosują wnioskowanie w przód.
3. „Ramy" (frames) i sieci semantyczne – wiedza dana jest w postaci strukturalnej i zorientowanej obiektowo.

Systemy **dowodzenia twierdzeń** (ang. *theorem provers*) (np. AURA, OTTER) stosują wnioskowanie metodą rezolucji w logice predykatów i ich celem jest dowodzenia twierdzeń w zadaniach matematycznych oraz realizacja zapytań do bazy wiedzy.

Języki **programowania w logice**, (np. język **Prolog**), stosują zwykle wnioskowanie wstecz, nakładają ograniczenia na język predykatów i dysponują proceduralnymi elementami wejścia/wyjścia. Program w Prologu to uporządkowany ciąg formuł a dane to formuła celu (zapytanie). Wykonanie programu to realizacja wnioskowania wstecz z przeszukiwaniem według zasady „*przeszukiwanie w głąb*", i „*od lewej-do prawej*". Programowanie w Prologu ma zasadniczo charakter deklaratywny (kod programu jest logiczną specyfikacją problemu a jego wykonanie – wnioskowaniem logicznym). Jednak w celu zwiększenia efektywności występuje też wiele mechanizmów pozalogicznych: jest duży zbiór wbudowanych predykatów związanych z arytmetyką i wejściem/wyjściem – sprawdzenie poprawności tych predykatów polega na wykonaniu odpowiedniego kodu procedur a nie na wnioskowaniu logicznym; występuje też szereg funkcji systemowych i obsługujących bazę wiedzy.

Systemy **regułowe** (**produkcyjne)** (np. OPS5, CLIPS, SOAR) stosują *reguły* (implikacje wiążące warunki wykonania z opisami akcji) i są podstawą wielu systemów ekspertowych. Systemy produkcji realizują wnioskowanie w przód, używając bardzo ograniczonego języka. Typowy system produkcji składa się z: pamięci roboczej – zawiera pozytywne literały bez zmiennych (zdania atomowe); zbioru reguł – wyrażeń postaci:

$$P_1 \wedge ... \wedge P_n \Rightarrow akcja_1 \wedge ... \wedge akcja_m \,,$$

gdzie (P_1,...,P_n) są literałami (formułami atomowymi), a ($akcja_1$,..., $akcja_m$) są działaniami, które należy wykonać wtedy, jeśli wszystkie zdania P_i są prawdziwe (tzn. znajdują się w pamięci roboczej). Wśród akcji są operacje wprowadzania i usuwania do/z bazy wiedzy i operacje we/wy.

„**Ramy**" (ang. *frames*) i **sieci semantyczne** (np. SNePS, Netl, KL-ONE) są to etykietowane grafy o ściśle zdefiniowanych etykietach łuków. Obiekty i ich kategorie są węzłami grafu a ich relacje binarne reprezentują zależności typu: „*jest częścią*", „*jest specjalizacją*", „*jest instancją*". Każdy węzeł sieci, posiada nazwane atrybuty, gdzie każdy atrybut wiąże obiekt z jakimś termem (w szczególności ze stałą). Atrybuty kategorii ogólnej są dziedziczone przez jej specjalizacje. W zasadzie każda sieć semantyczna może być wyrażona w postaci zbioru formuł logiki predykatów. Jednak sieć semantyczna realizująca dziedziczenie z możliwością wystąpienia **wyjątków** jest

potencjalnie **niemonotoniczna** – dodanie nowej przesłanki może unieważnić dotychczas wyprowadzone formuły.

5.4. PROLOG

Język programowania logicznego umożliwia implementację logicznego systemu wnioskowania dzięki temu, że:

- posiada reprezentację logicznych formuł,
- umożliwia dołączanie informacji sterującej procesem wnioskowania,
- posiada mechanizm wnioskowania.

Typowym przedstawicielem takich języków jest PROLOG. Program w Prologu jest zbiorem klauzul Horna - czyli każda formuła jest postaci formuły atomowej lub implikacji, gdzie w poprzedniku występują dodatnie literały a następnik jest pojedynczym literałem. Dużymi literami oznacza się zmienne, małymi – stałe. Prawdziwość predykatów sprawdzana jest za pomocą dołączonego kodu programu a nie w wyniku wnioskowania.

W Prologu stosuje się zapis klauzul Horna w formacie implikacji pisanej w odwrotnej kolejności (następnik – nagłówek poprzedza poprzednika implikacji) a literały poprzednika są niejawnie połączone koniunkcją, czyli:

nagłówek **:-** literał$_1$, … literał$_n$.

Np.:

przestępca(X) **:-** *Amerykanin*(X), *broń*(Y), *sprzedaje*(X,Y,Z), *wrogi*(Z).

Każda klauzula Horna w Prologu jest jednej z postaci:

$\neg P_1 \vee ... \vee \neg P_n \vee Q$ (*równoważne:* $(P_1 \wedge ... \wedge P_n) \Rightarrow Q$)

Q (*formuła atomowa – literał*)

$\neg P_1 \vee ... \vee \neg P_n$ (*klauzula celu*)

$\perp$ (*klauzula pusta*)

W Prologu podane powyżej klauzule Horna zapisujemy w następującej postaci:

Q :- $P_1, ..., P_n$.

Q :- . (*formuła atomowa – literał*)

:- $P_1, ..., P_n$. (*klauzula celu*)

$\perp$. (*klauzula pusta*)

Klauzule Horna posiadają swoje **interpretacje w programie**:

- Klauzulę postaci Q :- $P_1, ..., P_n$. interpretujemy jako procedurę o nagłówku Q i treści $P_1, ..., P_n$. Wywołanie procedury Q sprowadza się do kolejnego wywołania procedur $P_1, ..., P_n$.
- Klauzulę Q :- . interpretujemy jako procedurę o pustej treści.
- Klauzulę :- $P_1, ..., P_n$. traktujemy jako dane programu.
- Klauzulę pustą traktujemy jako instrukcję *stop*.

W Prologu istnieje szereg wbudowanych predykatów operacji arytmetycznych.

Np. X **is** 4+3 – formuła jest prawdziwa, gdy zmienna X posiada wartość 7.

5 **is** X+Y – prawdziwość takiej formuły nie może zostać sprawdzona na gruncie wbudowanych operacji.

- Wbudowane predykaty „operacji specjalnych” (np. WE/WY, przypisania).

Przyjęte jest założenie o zupełności świata - symbol "negacji” nie jest używany do tworzenia zanegowanych formuł, lecz odpowiada operacji **not** („nieprawda, że w aktualnej interpretacji zachodzi dany fakt”).

Np. mając klauzulę: *alive*(X) :- **not** *dead*(X), konkretny fakt *alive*(*joe*) zajdzie wtedy, jeśli wykażemy, że *dead*(*joe*) jest nieprawdziwe.

Program w Prologu to uporządkowany ciąg procedur:

- Dane to klauzula celu.
- Wykonanie programu to realizacja wnioskowania wstecz (lub metody rezolucji) z przeszukiwaniem według zasady „przeszukiwanie w głąb”, i „od lewej-do prawej”.

Programowanie w Prologu ma zasadniczo charakter deklaratywny (kod programu jest logiczną specyfikacją problemu a jego wykonanie – wnioskowaniem logicznym). Jednak w celu zwiększenia efektywności występuje też wiele mechanizmów pozalogicznych :

- duży zbiór wbudowanych predykatów związanych z arytmetyką i wejściem/wyjściem – sprawdzenie poprawności tych predykatów polega na wykonaniu odpowiedniego kodu procedur a nie na wnioskowaniu logicznym;
- szereg funkcji systemowych i funkcji obsługujących bazę wiedzy.

Przykład zapytania
Niech append oznacza operację połączenia dwóch list wraz z podaniem wyniku połączenia. Wśród danych programu mogą wystąpić klauzule:

:- append([], Y, Y) (połączenie zbioru pustego i Y , co daje w wyniku Y)

append([X|L],Y,[X|Z]) :- append(L,Y,Z) (rozszerzenie łączonego zbioru zawsze odpowiednio rozszerza wynik).

- Zapytanie: append(A, B, [1,2]) ?
- Odpowiedź:

A=[] B=[1,2]

A=[1] B=[2]

A=[1,2] B=[]

5.5. System regułowy

System regułowy (nazywany też systemem produkcji) jest podstawą większości systemów ekspertowych spotykanych w praktyce. Taki system realizuje **wnioskowanie wprzód**. Typowy system regułowy (produkcji) składa się z:

- **pamięci *roboczej***, która zawiera pozytywne literały bez zmiennych (zdania atomowe);
- zbioru **reguł (produkcji),** czyli wyrażeń o postaci:

$$P_1 \wedge ... \wedge P_n \Rightarrow akcja_1 \wedge ... \wedge akcja_m ,$$

gdzie $P_1,...,P_n$ są literałami (formułami atomowymi), a $akcja_1,..., akcja_m$ są działaniami, które należy wykonać wtedy, jeśli wszystkie zdania P_i są prawdziwe (tzn. znajdują się w pamięci roboczej). W

takiej sytuacji regułę nazywamy **realizowalną**. Przykłady akcji - dodanie lub usunięcie elementu z pamięci roboczej.

Praca systemu regułowego składa się z ciągu **cykli**. Z kolei każdy cykl składa się z **trzech faz**:

1. **faza dopasowania** – poszukiwanie realizowalnych reguł;
2. **faza wyboru** – wybór reguł do wykonania i ustalenie ich kolejności;
3. **faza wykonania** – wykonanie wybranych reguł.

5.5.1. Faza dopasowania

Zasadniczym elementem **fazy dopasowywania** jest procedura **unifikacji**, ale w praktycznych systemach wymagana jest jej efektywna implementacja. Przykładem efektywnego algorytmu dopasowywania jest tzw. **algorytm sieci czasu rzeczywistego RT** (tzw. RT-sieć) w OPS-5. Ogranicza on złożoność unifikacji dzięki jednokrotnemu wydzieleniu wspólnych zależności wielu reguł oraz pamiętaniu wcześniej wykonanych unifikacji.

Przykład algortymu „sieć RT"

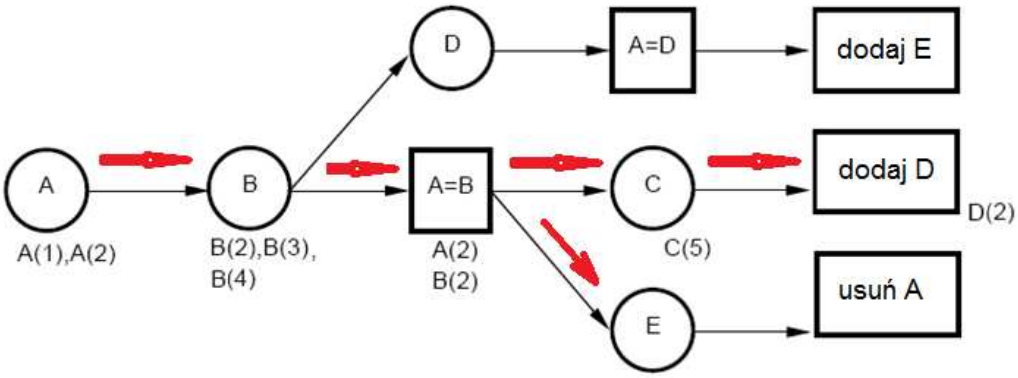
Załóżmy, że dane są następujące fakty (w pamięci *roboczej*) i reguły (w pamięci *długotrwałej*):

- Fakty: $\{ A(1), A(2), B(2), B(3), B(4), C(5) \}$
- Reguły: $A(x) \wedge B(x) \wedge C(y) \Rightarrow$ *dodaj* $D(x)$.

  $A(x) \wedge B(y) \wedge D(x) \Rightarrow$ *dodaj* $E(x)$.

  $A(x) \wedge B(x) \wedge E(z) \Rightarrow$ *usuń* $A(x)$.

Przed wykonaniem wnioskowania wykonywana jest konwersja reguł do postaci sieci obliczeń: okręgi to odwołania do pamięci roboczej, kwadraty to unifikacje, prostokąty to wyprowadzane akcje (rys. 5.3).



Rys. 5.3: Przykład wnioskowania w systemie regułowym "sieci RT"

5.5.2. Faza wyboru akcji

Faza wyboru – niektóre systemy wykonują wszystkie realizowalne działania, ale inne systemy wykonują tylko niektóre z nich. W takiej sytuacji są możliwe różne strategie wyboru:

- Strategia (1) - preferujemy reguły, które odnoszą się do ostatnio tworzonych elementów pamięci roboczej;
- Strategia (2) – preferujemy reguły bardziej „specyficzne".

Np. z poniższych reguł należy wybrać drugą z nich:

$Ssak(x) \Rightarrow$ dodaj $Nogi(x,4)$.

$Ssak(x) \wedge Człowiek(x) \Rightarrow$ dodaj $Nogi(x,2)$.

- Strategia (3) – preferujemy działania, którym nadano wyższy priorytet.

Np. z poniższych reguł zapewne preferujemy drugą z nich:

$P(x) \Rightarrow Akcja(Odkurzanie(x))$.

$R(x) \Rightarrow Akcja(\text{Ewakuacja})$.

5.6. Sieci semantyczne (ramy)

5.6.1. Sieć semantyczna

Sieć semantyczna to reprezentacja wiedzy w postaci **grafu**, przedstawiającego **kategorie obiektów** (lub pojedyncze obiekty) i **relacje** pomiędzy nimi. Podstawowe **relacje** to:

- *Subset* – relacja pomiędzy kategorią ogólną a specjalizowaną (relacja dziedziczenia);
- *Part* – relacja części do całości;
- *Member* (lub Instance) – relacja instancji (obiektu) do jej kategorii.

Każdy węzeł sieci (kategoria obiektów lub obiekt) posiada **nazwane atrybuty**, gdzie każdy atrybut wiąże obiekt z jakimś termem (w szczególności ze stałą). Atrybuty kategorii ogólnej są dziedziczone przez jej specjalizacje.

5.6.2. Logika a sieć semantyczna

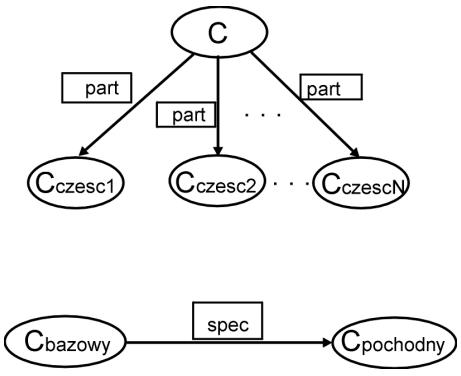
W zasadzie każda sieć semantyczna może być wyrażona w postaci zbioru formuł logiki 1 rzędu (rys. 5.4).

- Zależność „zbiór części –part_of –> całość” jest interpretowana jako formuła implikacji:

  $C_{część1} \wedge C_{część2} \wedge ... \wedge C_{częśćN} \Rightarrow C$

- Podobnie, zależność „koncept_bazowy –spec-> koncept_pochodny”:

  $C_{bazowy} \Rightarrow C_{pochodny}$



Rys. 5.4

Niech atrybut *R* dla węzła *A* wyznacza wartość *B*. **Semantyka atrybutu**: atrybut *R* może wyrażać relację 2-argumentową będącą 1 z 3 rodzajów:

- relacja zachodzi pomiędzy 2 obiektami - kategoriami: $R(A,B)$;
- relacja zachodzi pomiędzy każdym obiektem kategorii *A* i obiektem *B*: $\forall x \; x \in A \Rightarrow R(x,B)$;
- relacja zachodzi pomiędzy każdym obiektem kategorii *A* i pewnym elementem kategorii *B*: $\forall x \; \exists y \; x \in A \Rightarrow y \in B \wedge R(x,y)$.

5.6.3. Wyjątki

Jak wyrazić **wyjątki** dla relacji R zachodzące dla instancji kategorii A lub dla jej kategorii specjalizowanych?

Niech odpowiednikiem atrybutu – pojęcia stosowanego w sieci semantycznej – będzie w logice *L1R* formuła ze specjalnie wprowadzonym predykatem Rel:

$$Rel(R, A, B).$$

Interpretacja formuły atomowej, $Rel(R, A, B)$: mówimy, że B jest wartością domyślną atrybutu R dla instancji kategorii A.

Kolejnym krokiem ku reprezentacji wyjątków jest **reifikacja,** czyli proces zamiany relacji na obiekt. Relacja R staje się obiektem a przestaje być predykatem, tzn. że $R(A, B)$ będzie oznaczać obiekt.

- Teraz $Rel(R, A, B)$ redukuje się do formuły atomowej.

Dodamy predykat, $Val(R, x, B)$, który oznacza, że formuła $Rel(R, x, B)$ znajduje się w bazie wiedzy. Dodamy też formułę: $\mathrm{Jest}(R, x, B)$, który oznacza, że $Rel(R,x,B)$ wynika z bazy wiedzy. Zdefiniujemy ten predykat jako:

$$\forall r,x,b\ Jest(r,x,b) \Leftrightarrow [Val(r,x,b) \vee \exists p\ \ x \in p \wedge Rel(r,p,b)\ \ \wedge \neg InterwenRel(x,p,r)].$$

$$\forall x,p,r\ InterwenRel(x,p,r) \Leftrightarrow [\exists i\ Interwen(x,i,p) \wedge \exists b'\ Rel(r,i,b')].$$

$$\forall x,i,p\ Interwen(x,i,p) \Leftrightarrow (x \in i) \wedge (i \subset p).$$

5.6.4. Logika niemonotoniczna

Powinniśmy przekazać w bazie wiedzy informację o zupełności relacji Rel i Val.

Np.: $\forall r,a,b\ Rel(r,a,b) \Leftrightarrow [r,a,b] \in \{[Żywy, Zwierzę, T], \ldots\}$.

$$\forall r,a,b\ Val(r,a,b) \Leftrightarrow [r,a,b] \in \{[Przyjaciel, Kot, Bill], \ldots\}.$$

Jednak sieć semantyczna realizująca **dziedziczenie z możliwością wystąpienia wyjątków** jest potencjalnie **niemonotoniczna** – dodanie nowej przesłanki może unieważnić dotychczas wyprowadzone formuły.

Np. dla rozważanej sieci dodanie $Rel(Nogi, Koty, 3)$ unieważnia wyprowadzony poprzednio fakt, że $Bill$ ma 4 nogi.

Problem: logika klasyczna jest **ekstensjonalna** tzn. wartość wyrażenia jest funkcją wartości jego podwyrażeń.

Prowadzi to do następującego wynikania:

z faktu $(A = B)$ wynika, że $Zna(Agent, A) \Leftrightarrow Zna(Agent, B)$.

Taka cecha może prowadzić do generowania zupełnie niepotrzebnych formuł.

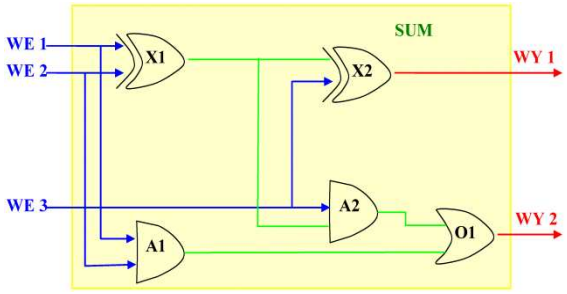
5.7. Pytania

1. Czym zajmuje się inżynieria wiedzy?
2. Wymienić podstawowe kategorie ontologiczne.
3. Omówić system PROLOG.
4. Omówić przykładowy system regułowy.
5. Przedstawić sieć semantyczną i jej zasady wnioskowania

5.8. Zadania

Zad. 5.1

Zaprojektować bazę wiedzy dla reprezentacji układu cyfrowego sumatora jedno-bitowego (rys. 5.5).



Rys. 5.5

Zad. 5.2

Zaimplementować w języku **Prolog** predykat sorted(L), który jest prawdziwy, jeśli lista L jest posortowana.

B) Zaimplementować w języku **Prolog** predykat merge(L1,L2,L3), który jest prawdziwy wówczas, gdy listy L1, L2 i L3 są posortowane, przy czym lista L3 zawiera wszystkie elementy zawarte w L1 i L2. Zaobserwować wyniki działania predykatu przy wywołaniu:

1) merge([1,3,5],[2,4],L).
2) merge([1,3,5],L,[1,2,3,4,5]).
3) merge(L1,L2,[1,2,3,4,5]).

Część II: Przeszukiwanie

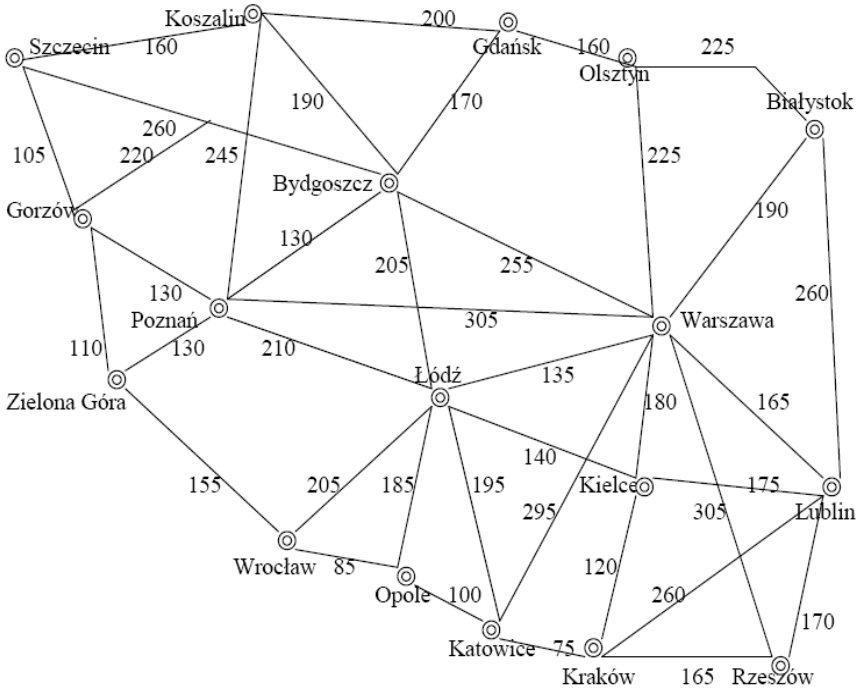
6. Przeszukiwanie przestrzeni stanów

6.1. Przestrzeń stanów

W tym rozdziale omówimy metodykę „przeszukiwania przestrzeni stanów" jako generalny (abstrakcyjny) sposób działania agenta realizującego cel. O ile tylko uda nam się wyrazić konkretny problem z konkretnej dziedziny w terminach tej metodyki to będziemy mogli natychmiast skorzystać z istniejącego sposobu rozwiązania problemu.

6.1.1. Przykład

Dany jest przykładowy problem: „przejazd ze Szczecina do Krakowa". Załóżmy, że aktualnie jesteśmy w Szczecinie i chcemy pojechać do Krakowa transportem drogowym. Celem działania jest: *dotrzeć do Krakowa*. Kluczową sprawą jest wyrażenie możliwych rozwiązań problemu w postaci **grafu stanów**, gdzie pojedynczy stan oznacza „*agent przebywa w danym mieście*". Dla uproszczenia wyróżnimy tylko większe miasta w Polsce (rys. 6.1). Możliwe **akcje** (operatory w przestrzeni stanów) oznaczają: „*przejazd z miasta A do miasta B*". Rozwiązaniem będzie każda ścieżka w grafie stanów prowadząca od stanu początkowego do stanu końcowego. W tym ostatnim spełniony jest **warunek zatrzymania** działania. Rozwiązaniem może być np. sekwencja akcji dla przejazdu: (Szczecin, Bydgoszcz, Łódź, Katowice, Kraków).



Rys. 6.1: Przykład grafu stanów dla problemu „przejazdu z miasta A do miasta B".

6.1.2. Problem przeszukiwania

Teraz podamy definicję **problemu przeszukiwania**. Problem wyrażony jest w postaci 4 pojęć (rodzajów danych):

1. Zbiór stanów S z wyróżnionym stanem początkowym; np. Szczecin $\in$ S, o znaczeniu: „*agent jest w Szczecinie*" ;
2. Akcje (operacje w przestrzeni stanów), A = $\{a_1, a_2, \ldots a_n\}$, i funkcja następnika stanu:

$[(stan\_we, a) \rightarrow stan\_wy] \in S \times A \times S$

Np.: (Szczecin, „*ze Szczecina do Bydgoszczy*”) → Bydgoszcz

3. Warunek osiągnięcia celu T, który jest spełniony w każdym stanie końcowym. Może on bezpośrednio referować stan, np., T: s = „*Kraków*", lub badać własność stanu, np. T: *Szachmat*(s) = true .
4. Koszt każdej akcji i sposób obliczenia kosztu rozwiązania (najczęściej jest to suma kosztów akcji tworzących rozwiązanie); np., suma odległości w km pomiędzy miastami, liczba wykonanych akcji, itp.

   Niech, $c(x, a, y)$, oznacza koszt akcji. W algorytmach realizujących różne strategie przeszukiwania zazwyczaj postuluje się, aby koszt akcji był nieujemny, $c \geq 0$.

Rozwiązaniem problemu jest sekwencja akcji (ścieżka w przestrzeni stanów) prowadząca od stanu początkowego do końcowego.

Uwaga: przedstawiliśmy tu definicję problemu **jednostanowego**, w którym każdy węzeł drzewa przeszukiwania odpowiada jednemu stanowi w przestrzeni problemu. Taki przypadek zachodzi wtedy, gdy środowisko działania agenta jest w pełni obserwowalne. W sytuacji, gdy środowisko nie jest w pełni obserwowalne mamy do czynienia z problemem wielostanowym – z uwagi na niepewność w jakim stanie znajduje się środowisko jeden węzeł drzewa przeszukiwania odpowiada wielu stanom problemu.

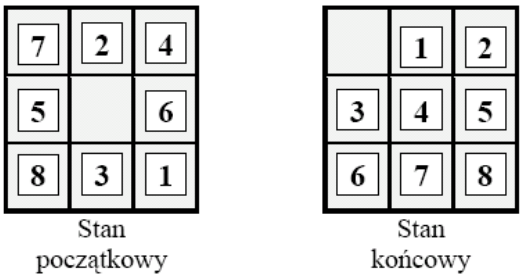
Wybór przestrzeni reprezentacji problemu

Rzeczywisty problem (świat) jest bardzo złożony a podana powyżej definicja problemu przeszukiwania obejmuje jedynie niezbędne elementy opisu tego świata. Podczas projektowania systemu sztucznej inteligencji trzeba stworzyć uproszczony (abstrakcyjny) model świata, w którym:

- Abstrakcyjny stan odpowiada wielu sytuacjom rzeczywistym;
- Abstrakcyjna akcja odpowiada wielu rzeczywistym akcjom; np., „(Szczecin, „przejazd”) → Bydgoszcz" reprezentuje zbiór możliwych dróg, objazdów, miejsc odpoczynku, itd.
- Abstrakcyjne rozwiązanie odpowiada zbiorowi rzeczywistych dróg, które w rzeczywistym świecie prowadzą do celu.

Tym samym każde z abstrakcyjnych pojęć stanowi zwykle uproszczenie oryginalnego pojęcia w rzeczywistym świecie.

Przykład. Definicja problemu przeszukiwania dla świata „8-puzzli” (rys. 6.2).



Rys. 6.2: Przykład stanów dla problemu „8-puzzli”.

Stany problemu reprezentują konfiguracje 8 numerowanych kafelków (płytek) w kwadratowym obszarze o rozmiarze 3×3. Pojedyncza akcja polega na „przemieszczeniu” pustego miejsca w lewo, prawo, na górę lub na dół (dualnie: jest to przesunięcie kafelka sąsiadującego z pustym miejscem). Warunek zatrzymania (stopu) jest podany jawnie w postaci stan końcowego, w którym numery kafelków uporządkowane są w kolejności rosnącej. Koszt każdej akcji wynosi 1.

Rozmiary różnych wersji „świata puzzli” mogą być dowolnie duże (oznaczmy rozmiar przez N). Problem polega na optymalnym (w sensie minimalnego kosztu potrzebnych akcji) uporządkowaniu „świata N-puzzli”. Rozwiązanie takiego problemu jest NP-trudne, czyli o złożoności obliczeniowej

powyżej wielomianowej względem N. Z naszego punktu widzenia problem „świata N-puzzli" stanowi dogodną ilustrację dla porównywania różnych **strategii przeszukiwania** przestrzeni stanów.

6.1.3. Przeszukiwanie drzewa

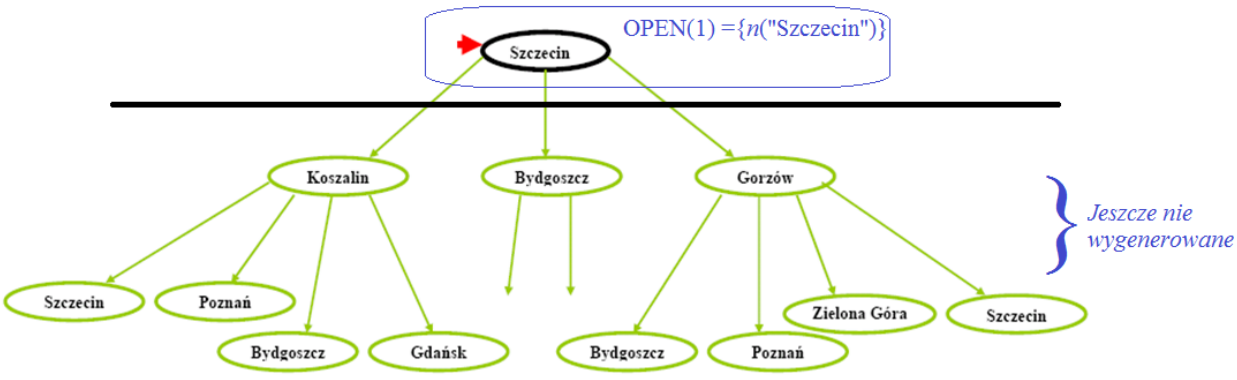
W zasadzie każdą strategię przeszukiwania dyskretnej przestrzeni problemu możemy wyrazić jako wizytowanie (przejście) pewnego **drzewa** lub **grafu**. Oczywiście struktura drzewa jest uproszczoną formą grafu. Dlatego najpierw ograniczymy nasze rozważania dotyczące strategii przeszukiwania do postaci przejścia po drzewie a następnie rozszerzymy je o dodatkowe elementy wymagane przy przejściu grafu.

Podczas przeszukiwania przestrzeni reprezentujemy wyniki częściowe w postaci węzłów **drzewa przeszukiwania**, a alternatywne akcje przeprowadzają aktualny węzeł w jeden z możliwych węzłów-następników tego drzewa. Rozwijamy drzewo przeszukiwania począwszy od korzenia, reprezentującego *stan początkowy*, poprzez węzły pośrednie do liści drzewa, z których przynajmniej jeden reprezentuje *stan końcowy,* czyli stanowi cel przeszukiwania.

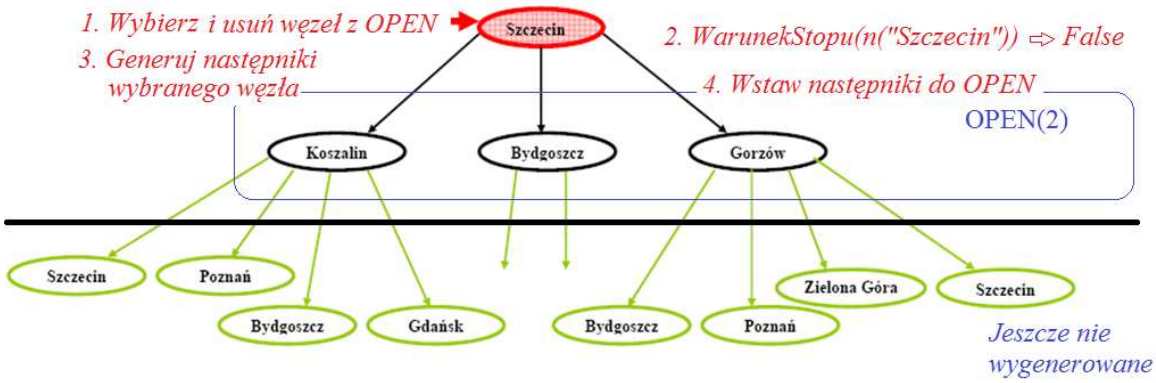
W problemie „jedno-stanowym" *węzeł* drzewa przeszukiwania odpowiada pojedynczemu stanowi problemu. Aktualny zbiór węzłów-liści, posiadających jeszcze niewizytowane następniki, to zbiór węzłów gotowych do rozwinięcia, utożsamiany z tzw. **skrajem** drzewa (często w algorytmach przeszukiwania nazywany zbiorem OPEN). Sposób porządkowania węzłów w skraju i tym samym sposób wyboru następnego rozwijanego węzła wyraża określoną strategię przeszukiwania.

Przykład (rys. 6.3 – 6.5)

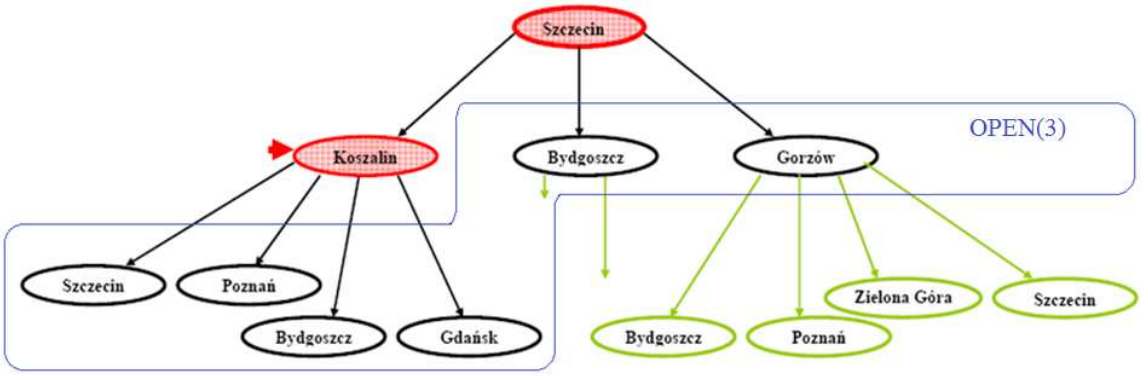
Początkowe drzewo decyzyjne dla problemu „powrót ze Szczecina do Krakowa" (rys. 6.3). Węzeł początkowy o etykiecie „Szczecin" reprezentuje sytuację (jest to także jeden stan problemu) „*agent jest w Szczecinie*". Jest to na początku jedyny węzeł w skraju, który jest w pierwszym kroku wybierany do rozszerzenia. Pozostałe widoczne węzły reprezentują możliwe następniki, które zostaną wygenerowane na podstawie opisu problemu (na podstawie grafu problemu) z chwilą wykonania określonej akcji. W pierwszym kroku generowane są następniki węzła „Szczecin", czyli węzły: Koszalin, Bydgoszcz, Gorzów (rys. 6.4.). Załóżmy, że w kolejnym kroku, zgodnie z przyjętą strategią przeszukiwania, wybierany jest węzeł „Koszalin". Nie spełnia on warunku celu więc generowane są jego następniki reprezentujące stany: Szczecin, Poznań, Bydgoszcz, Gdańsk (rys. 6.5). W kolejnych krokach postępujemy podobnie, wybierając węzły znajdujące się w skraju drzewa, sprawdzając dla nich warunek zatrzymania i w przypadku niespełnienia warunku – generując następników węzła.



Rys. 6.3: Przykład początkowego drzewa przeszukiwania dla problemu „powrót ze Szczecina do Krakowa".



Rys. 6.4: Postać drzewa przeszukiwania po wybraniu węzła „Szczecin”.

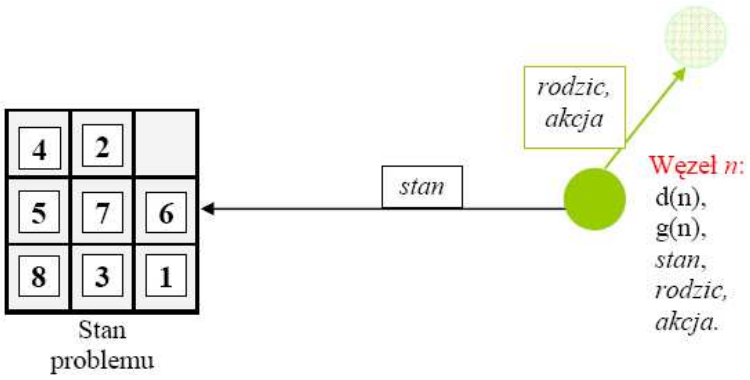


Rys. 6.5: Postać drzewa przeszukiwania po wybraniu węzła „Koszalin”.

Zauważmy, ze w powyższym przykładzie możemy wielokrotnie generować węzły reprezentujące ***ten sam*** stan problemu. Nie jest to sytuacja pożądana, gdyż może ona prowadzić do zamkniętych cykli (powtarzanie stanu na tej samej ścieżce), a te z kolei mogą być powtarzane nieskończenie wiele razy, co prowadzi do nieskończenie długiej ścieżki. Dla takich problemów będziemy musieli rozszerzyć problem przeszukiwania drzewa do problemu przeszukiwania grafu.

Wyjaśnijmy jeszcze *różnicę* pomiędzy **węzłem drzewa** przeszukiwania a **stanem** problemu. Stan jest abstrakcyjną reprezentacją fizycznej konfiguracji świata. Węzeł n jest strukturą danych i elementem drzewa przeszukiwania. Dla przykładu węzeł może zawierać (rys. 6.6):

- odniesienie do stanu lub zbioru stanów problemu,
- wskaźnik do węzła rodzica,
- akcję, wykonaną w sytuacji reprezentowanej węzłem rodzica,
- koszt sekwencji akcji prowadzącej do tego węzła $g(n)$ i głębokość węzła $d(n)$.



Rys. 6.6: Przykład struktury węzła w drzewie przeszukiwania.

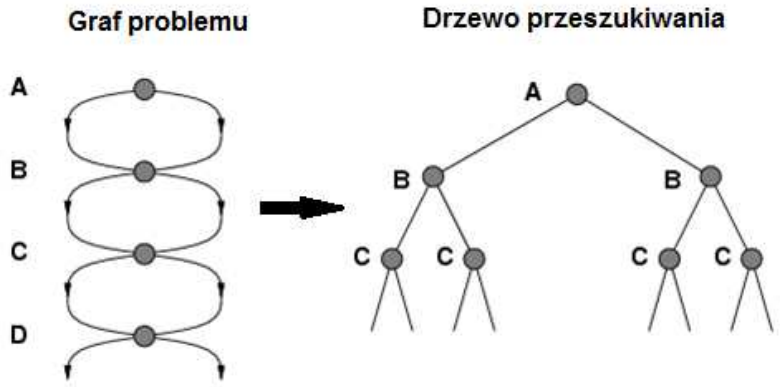
6.1.4. „Przeszukiwanie grafu”

Jak zauważyliśmy we wcześniejszym przykładzie w procesie przeszukiwania drzewa może dochodzić do sytuacji ponownego generowania węzłów reprezentujących te same stany problemu. Jest to spowodowane tym, że przestrzeń problem przyjmuje postać grafu (istnieją różne ścieżki łączące te same dwa stany) lub też istnieją dwukierunkowe łuki. Wyróżnimy dwa przypadki generowania równoważnych węzłów (w sensie reprezentowania tego samego stanu problemu):

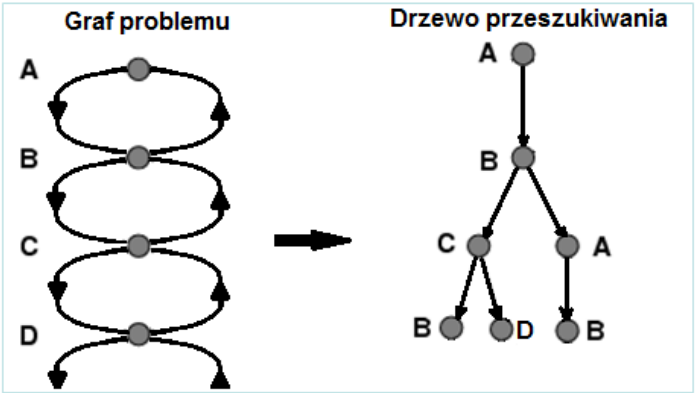
1. określony stan jest wielokrotnie rozwijany na różnych ścieżkach drzewa przeszukiwania (rys. 6.7) lub
2. stan jest powtórnie „wizytowany” na tej samej ścieżce drzewa przeszukiwania (rys. 6.8).

Pierwszy przypadek może prowadzić do niepotrzebnego rozwijania nadmiarowych ścieżek i wpływać negatywnie na efektywność strategii przeszukiwania.

Drugi przypadek odpowiada powstaniu pętli na ścieżce rozwiązania, co może prowadzić do ścieżki o nieskończonej długości i braku rozwiązania. Brak wykrywania takich sytuacji może prowadzić do wielokrotnego powtarzania tej samej sekwencji akcji, czasem nieskończoną ilość razy.



Rys. 6.7: Wielokrotne węzły na różnych ścieżkach drzewa przeszukiwania



Rys. 6.8 Powtórny węzeł na tej samej ścieżce (pętla)

Powyższe sytuacje wymagają modyfikacji bazowego mechanizmu przeszukiwania drzewa. Dla przykładu, przeciwdziałamy obu powyższym niepożądanym przypadkom dokonując następujących zmian i uzupełnień:

1. należy zapamiętać wszystkie wcześniej rozwijane węzły (w tym celu wprowadzamy zbiór węzłów CLOSED),
2. każdy nowo generowany węzeł jest porównywany z węzłami znajdującymi się w zbiorach OPEN i CLOSED; po ewentualnym stwierdzeniu identyczności stanów reprezentowanych dwoma różnymi węzłami, jeden z węzłów (domyślnie ten „gorszy”) jest eliminowany.

6.2. Strategie ślepego przeszukiwania

Ślepe przeszukiwanie (ang. *uninformed search*) wykorzystuje jedynie informację zawartą w sformułowaniu problemu.

- Przeszukiwanie wszerz (*breadth-first search, BFS*) - skraj drzewa jest kolejką FIFO (first-in first-out).
- Przeszukiwanie z jednolitą funkcją kosztu (*uniform-cost search*) - węzły uporządkowane są w skraju według niemalejących sumarycznych kosztów dotychczasowych akcji prowadzących do danego węzła.
- Przeszukiwanie w głąb (*depth-first search, DFS*) - skraj jest stosem LIFO (last-in first-out).
- Przeszukiwanie z ograniczoną głębokością DLS (*depth-limited search*) - tak jak w głąb do zadanego ograniczenia l, ale węzły na poziomie ograniczenia nie mają już następników.
- Iteracyjne pogłębianie IDS (*iterative deepening search*) - kolejne, niezależne od siebie wykonywanie przeszukiwań DLS dla coraz większych wartości ograniczenia głębi (l=0,1,2,...), aż do momentu znalezienia celu.

6.2.1. Kryteria oceny strategii przeszukiwania

Przyjęta strategia przeszukiwania drzewa wyraża się w sposobie wyboru kolejno rozwijanych (wizytowanych) węzłów. Oczywiście dobrze byłoby móc porównać działanie różnych strategii. Podamy tu cztery podstawowe **kryteria** oceny strategii.

1. **Zupełność**;

   Zupełność strategii przeszukiwania oznacza, że jeżeli istnieje rozwiązanie problemu, to zostanie ono zawsze znalezione.

2. **Złożoność czasowa**: czas przeszukiwania mierzony całkowitą liczbą węzłów wizytowanych podczas przeszukiwania;
3. **Złożoność pamięciowa**: maksymalna liczba węzłów jednocześnie rezydujących w pamięci programu przeszukiwania;
4. **Optymalność**;

   Strategia optymalna to taka, która zawsze znajduje najlepsze rozwiązanie.

W przypadku kryteriów złożoności czasowej i pamięciowej interesują nas nie tyle złożoności dla konkretnego problemu, wyznaczone po procesie przeszukiwania, lecz oczekiwane złożoności, szacowane dla danej strategii na podstawie znajomości rozmiaru problemu. Przyjęło się wyrażać rozmiar problemu za pomocą następujących parametrów:

- b: maksymalne rozgałęzienia drzewa przeszukiwania,
- d: głębokość, na której znajduje się najtańsze rozwiązanie,
- m: maksymalna głębokość drzewa przeszukiwania.

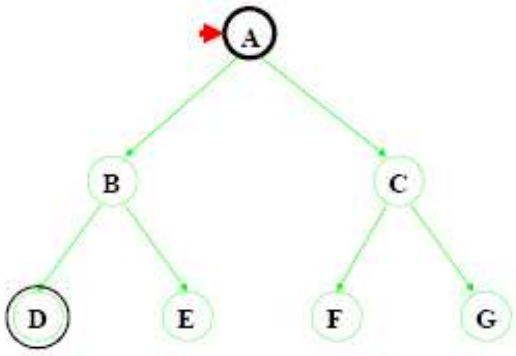
6.2.2. Przeszukiwanie wszerz – BFS

W tej strategii przeszukiwania drzewa rozwijany jest zawsze najpłytszy dotąd nie rozwinięty węzeł.

Implementacja strategii przeszukiwania wszerz polega na reprezentowaniu aktualnego skraju drzewa (tzw. zbiór OPEN) w postaci kolejki FIFO – nowo dodawane węzły-następniki ustawiane są zawsze na końcu kolejki a pobieranie węzłów (w celu rozwinięcia) ma miejsce na początku kolejki.

Przykład.

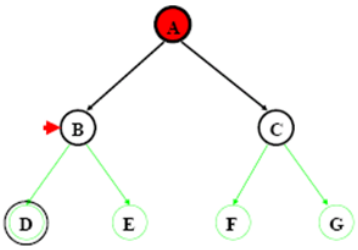
Dane jest drzewo przeszukiwania o trzech poziomach węzłów i węźle początkowym A (rys. 6.9).



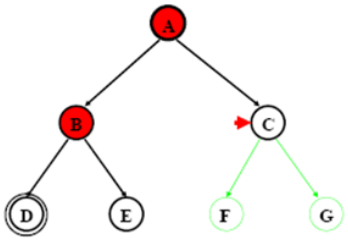
Rys. 6.9

Iteracja 1: Skraj(1) = {A}, wybierany jest węzeł A, generowane są jego następniki B, C.

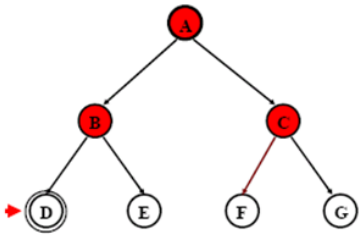
Iteracja 2: Skraj(2) = {B, C}, wybierany jest węzeł B, generowane są jego następniki D i E.



Iteracja 3: Skraj(3) = {C, D, E}, wybierany jest węzeł C, generowane są jego następniki F, G



Iteracja 4: Skraj(4) = { D,E,F,G}, wybierany jest węzeł D, nie są generowane żadne następniki, jeśli D spełnia warunek stopu to KONIEC.



Podsumowanie: rozwijane węzły A, B, C, D.

Rys. 6.10: Kolejne kroki przeszukiwania wszerz drzewa z rys. 6.9

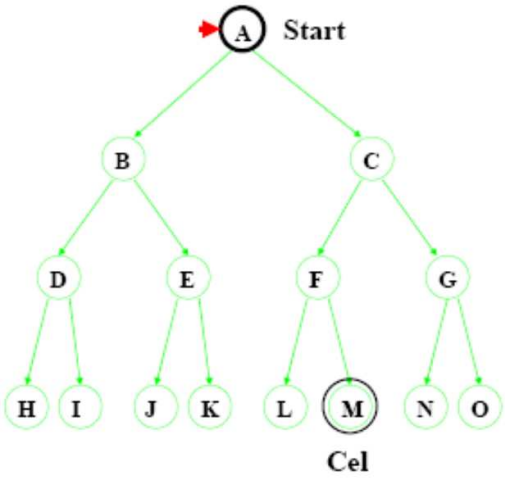
Węzeł A jest węzłem początkowym, a węzeł D – końcowym. Kolejność wybieranych (rozwijanych) węzłów w strategii przeszukiwania wszerz pokazano na rys. 6.10. Ostatecznie kolejność ta wynosi: A, B, C, D.

6.2.3. Przeszukiwanie w głąb – DFS

Zasada strategii przeszukiwania w głąb: rozwijany jest najgłębszy, dotąd nie rozwinięty węzeł. Implementacja *skraju drzewa* (czyli tzw. listy OPEN) ma postać stosu LIFO; nowe następniki ustawiane są na początku stosu i są rozwijane w pierwszej kolejności. Taka kolejność rozwijania węzłów odpowiada, np. lewostronnemu obejściu drzewa.

Przykład

Dane jest drzewo przeszukiwania o czterech poziomach węzłów i węźle początkowym A (rys. 6.11). Strategia przeszukiwania w głąb znajduje ścieżkę od węzła A (start, początkowy) do węzła docelowego M, rozwijając (wybierając „po drodze") następujące węzły (rys. 6.12): A, B, D, H, I, E, J, K, C, F, L, M.



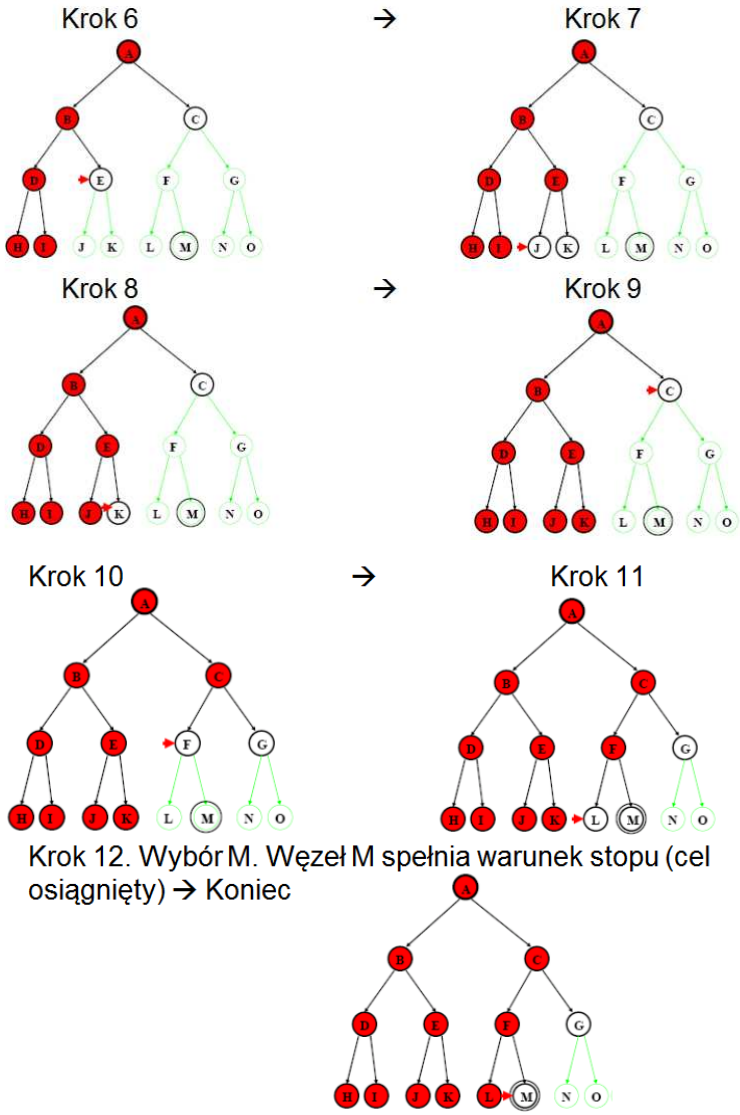
Rys. 6.11 Drzewo przeszukiwania ilustrujące strategię przeszukiwania w głąb

Krok 1: Skraj(1)={A}, wybór A.

Krok 2: Skraj(2)={B,C}, wybór B. →Krok 3: Skraj(3)={D,E,C}, wybór D.

Krok 4: Skraj(4)={H,I,E,C}, wybór H. → Krok 5: skraj(5)={I,E,C}, wybór I.

Rys. 6.12: Początek przeszukiwania w głąb drzewa z rys. 6.11



Rys. 6.12 (c.d.) Kontynuacja przeszukiwania drzewa z rys. 6.11 i zakończenie przeszukiwania

6.2.4. Przeszukiwanie z jednolitą funkcją kosztu - UCS

Strategia **przeszukiwania z jednolitą funkcją kosztu**: zawsze rozwija dotąd nie rozwinięty węzeł o najniższym koszcie z dotychczasowych. *Skraj* drzewa jest kolejką uporządkowaną według kosztu ścieżki (sekwencji akcji) prowadzącej do danego węzła. Ta strategia jest równoważna przeszukiwaniu wszerz, jeżeli koszty wszystkich akcji są równe.

Własności.

- Zupełność: tak, jeżeli koszt każdej akcji ≥ ε, gdzie ε ≥ 0.
- Złożoność obliczeniowa: liczba węzłów o koszcie $g(n)$ ≤ koszt optymalnego rozwiązania, $O(b^{(C^*/\varepsilon)})$, gdzie C^* to koszt optymalnego rozwiązania.
- Złożoność pamięciowa: liczba węzłów o koszcie $g(n)$ ≤ koszt optymalnego rozwiązania: $O(b^{(C^*/\varepsilon)})$.
- Optymalność: tak, w sensie minimalizacji kosztu, gdyż węzły rozwijane są zawsze w kolejności zwiększającego się kosztu $g(n)$.

6.2.5. Przeszukiwanie z ograniczoną głębokością – DLS

Jest to odmiana przeszukiwania w głąb z ograniczeniem nałożonym na głębokość węzła, co oznacza, że jeśli l jest ograniczeniem głębokości węzła to (z punktu widzenia strategii) węzły na głębokości l nie posiadają następników.

Strategia nie jest ani zupełna ani optymalna. Jeśli każde rozwiązanie jest dane na głębokości większej niż l to nie zostanie znalezione żadne rozwiązanie (przeszukiwanie niezupełne). Przeciwnie, jeśli istnieją rozwiązania na ścieżkach krótszych niż l to nie ma gwarancji, że znalezione rozwiązanie jest optymalne, nawet w sensie minimalnej długości ścieżki. Jest to efektem stosowania strategii przeszukiwania w głąb w ramach ograniczenia do głębokości l (strategia nieoptymalna).

6.2.6. Przeszukiwanie z iteracyjnym pogłębianiem – IDS

Strategia polega na iteracyjnym wywoływaniu przeszukiwania z ograniczoną głębokością dla różnych wartości ograniczenia głębokości l (l=0, 1, 2, …), tak długo, dopóki nie zostanie znalezione rozwiązanie, czyli ścieżka prowadząca do węzła końcowego (węzła spełniającego warunek stopu).

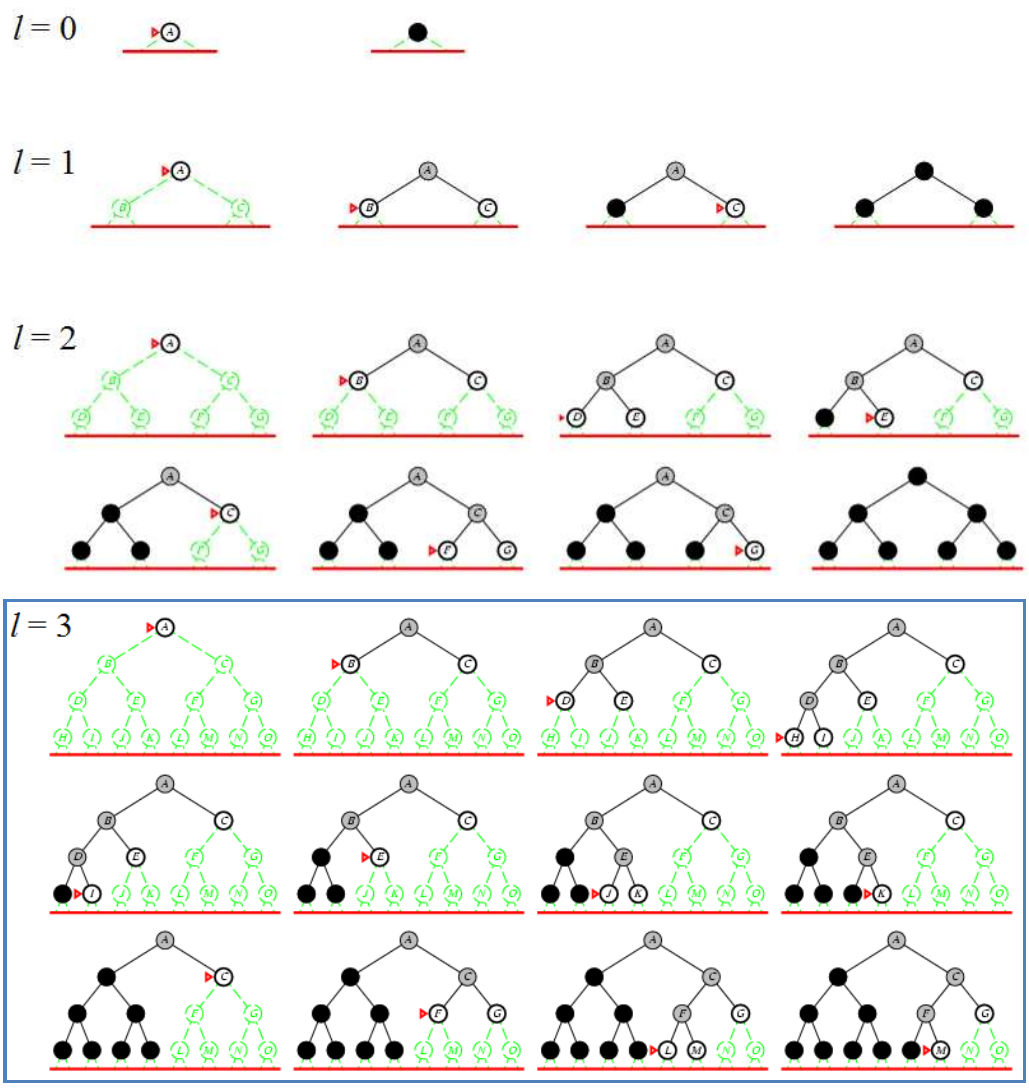
Funkcja implementująca strategię przeszukiwania IDS podana została w Tabeli 6-1.

Tab. 6-1 Strategia „iteracyjnego pogłębiania”.

```
function Iteracyjne_pogłębianie(problem), Wynik: ścieżka
{  for (int głębia=0; głębia < ∞ ; głębia ++) {
        wynik ←Przeszukiwanie_z_ograniczoną_głębokością(
                                problem, głębia)
      if (Typ(wynik) = Ścieżka_końcowa) then return wynik;
   }
}
```

Przykład

Dla drzewa przeszukiwania podanego na rys. 6.11 strategia IDS wymaga 4 iteracji (rys. 6.13).



Rys. 6.13: Kolejne iteracje w metodzie przeszukiwania IDS wykonywane dla drzewa z rys. 6.11

6.2.7. Porównanie strategii przeszukiwania

W tab. 6-2 określono zachowanie się strateg ślepego przeszukiwania w oparciu o wcześniej przyjęte kryteria. Dla przypomnienia parametry (symbole) oznaczają: b – stopień rozgałęzienia, d – długość ścieżki rozwiązania, m – maksymalna głębokość drzewa, l – ograniczenie głębokości drzewa, C^* - koszt optymalnego rozwiązania, ε - najmniejszy koszt akcji.

Tab. 6-2: Porównanie strategii ślepego przeszukiwania

Strategia: / **Kryterium**	„wszerz”	„z jednorodnym kosztem”	„w głąb”	„z ograniczoną głębokością”	„z iteracyjnym pogłębianiem”
Zupełny ?	Tak	Tak	Nie	Nie	Tak
Złożoność czasowa	$O(b^{d+1})$	$O(b^{\lceil C^*/\varepsilon \rceil})$	$O(b^m)$	$O(b^l)$	$O(b^d)$
Złożoność pamięciowa	$O(b^{d+1})$	$O(b^{\lceil C^*/\varepsilon \rceil})$	$O(bm)$	$O(bl)$	$O(bd)$
Optymalny ?	Tak	Tak	Nie	Nie	Tak

Iteracyjne pogłębianie (IDS) znajduje zawsze optymalną ścieżkę, posiada liniową złożoność pamięciową i nie potrzebuje dużo więcej czasu niż inne ślepe algorytmy przeszukiwania drzewa. Dlatego stanowi dogodną alternatywę dla **przeszukiwania z jednorodnym kosztem.**

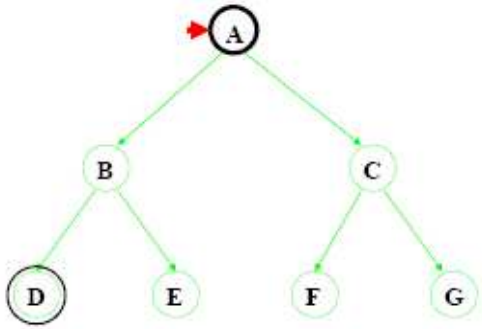
6.3. Pytania

1. Przedstawić wybrany problem w postaci problemu przeszukiwania.
2. Na czym polegają strategie ślepego przeszukiwania? Wymienić główne strategie i zilustrować je na przykładzie.
3. Przy pomocy jakich kryteriów porównujemy ze sobą strategie przeszukiwania?
4. Omówić problem przeszukiwania grafu.
5. Porównać ze sobą strategie ślepego przeszukiwania.

6.4. Zadania

Zad. 6.1

Wyjaśnić strategię **przeszukiwania wszerz** jako odmiany przeszukiwania **ślepego (niepoinformowanego)** stosowanego do rozwiązywania problemów. **Zilustrować** strategię **przeszukiwania wszerz** dla poniższego drzewa decyzyjnego, podając kolejność rozwijanych węzłów.



Zad. 6.2

Określić własności strategii przeszukiwania wszerz: zupełność, oczekiwana złożoność czasowa (obliczeniowa) i pamięciowa, optymalność.

Zad. 6.3

Określić własności strategii „z jednolitą funkcja kosztu”: zupełność, oczekiwana złożoność czasowa (obliczeniowa) i pamięciowa, optymalność.

Zad. 6.4

Określić własności strategii przeszukiwania w głąb: zupełność, oczekiwana złożoność czasowa (obliczeniowa) i pamięciowa, optymalność.

Zad. 6.5

Określić własności strategii iteracyjnego pogłębiania: zupełność, oczekiwana złożoność czasowa (obliczeniowa) i pamięciowa, optymalność.

Porównać oczekiwane liczby węzłów generowanych w DLS i IDS dla wybranych wartości parametrów.

7. Przeszukiwanie poinformowane

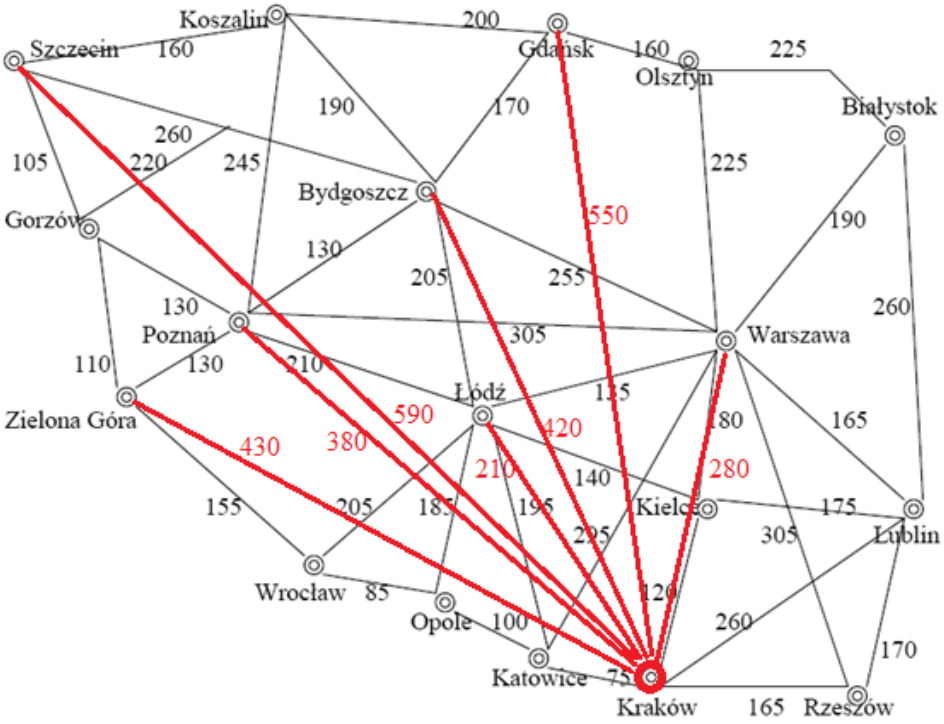
Strategia przeszukiwania jest wyznaczona sposobem wyboru kolejno rozwijanych węzłów w drzewie (grafie) przeszukiwania, tworzonym dla rozwiązania problemu. Istnieje kilka „ślepych" strategii poszukiwania, takich jak: przeszukiwanie wszerz, przeszukiwanie w głąb, przeszukiwanie z jednorodną funkcją kosztu, przeszukiwanie z ograniczoną głębokością i iteracyjnym pogłębianiem. Określenie „ślepy" oznacza w tym przypadku, że w żadnym przypadku na wybór następnego węzła nie wpływa informacja o stanie końcowym. Jest to ich zasadnicza niedogodność, która sprawia, że zwykle nie osiągają one wystarczającej efektywności obliczeniowej czy pamięciowej, a nawet niektóre z nich nie są optymalne.

Z analizy słabości strategii przeszukiwania ślepego wynika wniosek: należy użyć funkcji oceny dla węzłów drzewa przeszukiwania, która uwzględniałaby (abstrakcyjnie zdefiniowaną) „odległość" węzła od węzła dla stanu docelowego. Założenie o istnieniu takiej miary kosztów „resztkowych" na ścieżce rozwiązania pozwala zaproponować strategie tzw. przeszukiwania **poinformowanego**.

Oszacowanie kosztów resztkowych pozwoli ocenić na ile „obiecujący" z punktu widzenia celu jest dany węzeł i wybrać (rozwinąć) najbardziej „obiecujący" nierozwinięty węzeł.

Przykład (Rys. 7.1, Tab. 7-1)

Przykład składowej heurystycznej kosztu $h(n)$ dla problemu „powrót do Krakowa": odległość w linii prostej od danego miasta do Krakowa.



Rys. 7.1 Mapa rzeczywistych odległości pomiędzy stanami problem (wyróżnionymi miastami), a składowa heurystyczna dla wybranych stanów – odległość w linii prostej do stanu końcowego "Kraków".

Tab. 7-1. Przykład wartości składowej heurystycznej dla problemu „powrót do Krakowa”.

Białystok	440
Bydgoszcz	420
Gdańsk	550
Gorzów	500
Katowice	70
Kielce	110
Koszalin	580
Kraków	0
Lublin	230
Łódź	210
Opole	150
Poznań	380
Rzeszów	150
Olsztyn	460
Szczecin	590
Warszawa	280
Wrocław	240
Zielona Góra	430

7.1. Strategie „najpierw najlepszy” (*best first*)

W ogólności rozpatruje się funkcję oceny węzła (typowo jest to *koszt*) złożoną z 2 części:

$$f(n) = g(n) + h(n),$$

gdzie $g(n)$ oznacza koszt dotychczasowej ścieżki a $h(n)$ oznacza przewidywany koszt resztkowy pozostałej drogi z węzła n do celu. Implementacja każdej strategii, należącej do kategorii „najpierw najlepszy”, polega na porządkowaniu węzłów w skraju według zwiększającej się wartości kosztu węzła $f(n)$.

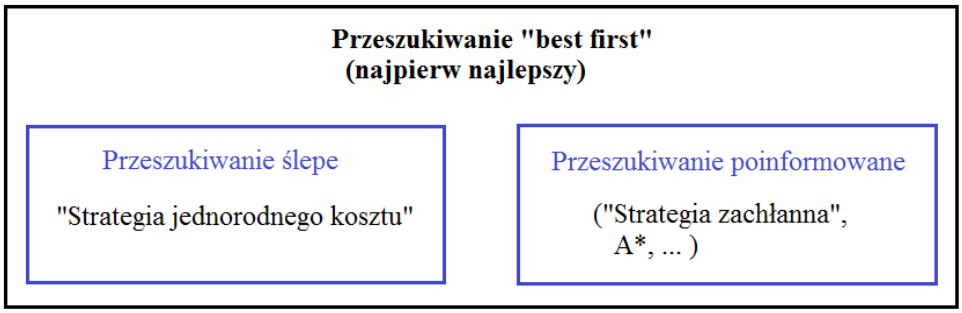
W zależności od postaci funkcji oceny rozpatrzymy trzy odmiany strategii „najpierw najlepszy”:

1. strategia jednorodnego kosztu: $f(n) = g(n)$,
2. strategia zachłanna („najbliższy celowi najpierw”): $f(n) = h(n)$,
3. przeszukiwanie A* : $f(n) = g(n) + h(n)$.

Strategia **jednorodnego kosztu** (wzgl. zysku), nazywana też strategią **równomiernego** kosztu (zysku), została omówiona w rozdziale 5, jako odmiana ślepego przeszukiwania. Posługuje się ona funkcją oceny węzła n, w której uwzględnia się jedynie koszty dotychczasowych akcji (na ścieżce od węzła początkowego do węzła n).
Nie zalicza się jej do strategii poinformowanego przeszukiwania. Przeszukiwanie poinformowane ma miejsce wtedy, gdy posługujemy się w funkcji oceny oszacowaniem kosztów resztkowych $h(n)$ (nazywanych **heurystyką**).

Uwaga (rys. 7.2)

Podkreślmy ponownie, że strategia **jednorodnego kosztu** należy do grupy strategii niepoinformowanych („ślepych”), gdyż nie uwzględnia ona składowej kosztu resztkowego, $h(n)$, (tzw. składowej *heurystycznej* funkcji kosztu). Pozostałe dwa przypadki przeszukiwania „*best first*” należą do strategii poinformowanych, gdyż korzystają z oszacowań kosztów resztkowych dla każdego węzła przestrzeni przeszukiwania. Zostaną one omówione w następnych punktach.



Rys. 7.2 Klasyfikacja strategii “najpierw najlepszy” (“*best first*”)

7.2. Strategia zachłanna („najbliższy celowi najpierw”)

W tej strategii funkcja oceny węzła przyjmuje postać:

$$f(n) = h(n) ,$$

czyli składa się wyłącznie ze składowej heurystycznej (oszacowania kosztów resztkowych przejścia z węzła n do celu). Zakładamy w niej, że najlepszy węzeł to ten, który jest „najbliższy od celu”. Dotychczas poniesione koszty dojścia do aktualnego węzła nie mają żadnego znaczenia dla decyzji wyboru. Szczegółowy algorytm implementacji przeszukiwania grafu według strategii zachłannej podano w Tabeli 7-2. Dzięki istnieniu zbioru CLOSED, do którego wstawiane są wizytowane węzły, i dzięki krokowi 6, uniemożliwiającemu wielokrotne wybieranie równoważnych węzłów (tego samego stanu problemu), algorytm realizuje przeszukiwanie **grafu**, a nie jedynie - drzewa.

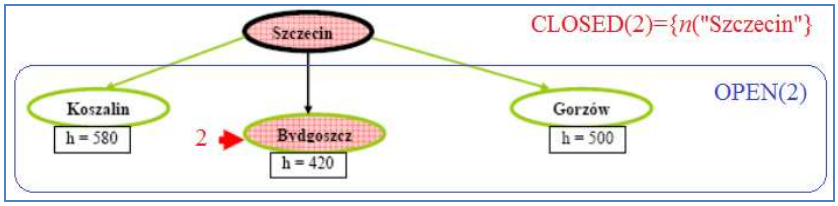
Tab. 7-2. Algorytm przeszukiwania grafu dla strategii „zachłannej”.

1		INIT: Pobierz węzeł startowy s i umieść go w zbiorze OPEN. Ustaw $f(s) = h(s)$
2		Pobierz z OPEN najlepszy węzeł n (o najmniejszym koszcie $f(n)$) i przenieś go do CLOSED
3		JEŚLI (n jest węzłem końcowym) TO zakończ i zwróć g(n) oraz całą ścieżkę od s do n.
4		Znajdź węzły następców n - niech będą nimi: $n_1' \ldots n_k'$.
5		Dla każdego z następców $n_1' \ldots n_k'$ wyznacz jego koszt $f_i' = h(n_i')$
6		Dla każdego z węzłów $n_1' \ldots n_k'$:
	a	JEŚLI (n_i' nie należy do zbioru OPEN ani do CLOSED) TO dodaj go do zbioru OPEN i ustaw: $f(n_i') = f_i'$.
	b	JEŚLI (n_i' należy do zbioru OPEN lub CLOSED i $f(n_i') > f_i'$) TO usuń dotychczasową ścieżkę od s do n_i' i ustaw $f(n_i') = f_i'$. Jeśli n_i' był w zbiorze CLOSED, to umieść go w zbiorze OPEN.
7		Powtórz od kroku 2.

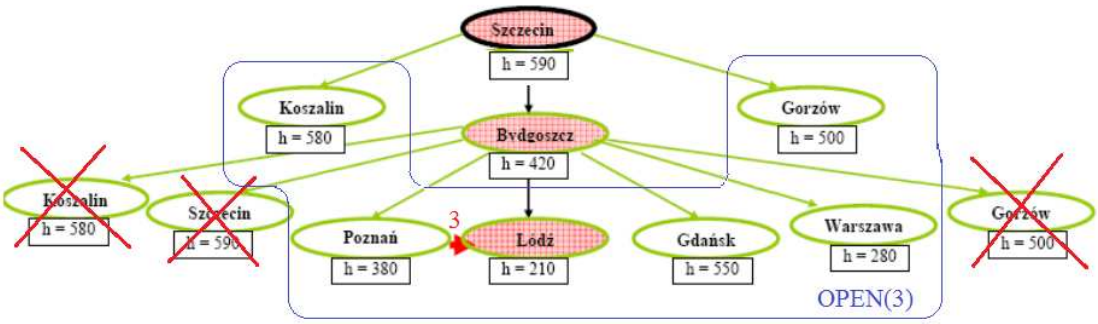
Przykład. Strategia „zachłanna” dla problemu „powrót ze Szczecina do Krakowa”. Załóżmy, że graf problemu podano na rys. 6.1, a liczby przy łukach reprezentują koszty akcji przejazdów z miasta do miasta. Dla strategii zachłannej nie mają one żadnego znaczenia. Kieruje się ona jedynie oszacowaniem kosztów resztkowych podanych w tab. 7-1 (ilustrowanych czerwonymi łukami

na rys. 7.1). Kroki przeszukiwania grafu, zdefiniowanego dla podanego problemu, wykonywane zgodnie z tą strategią, ilustrują rys. 7.3, 7.4.

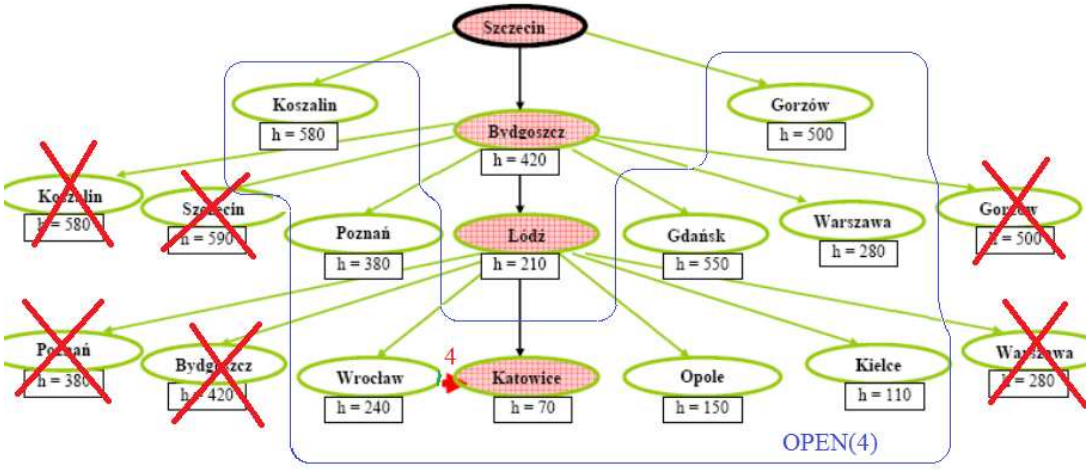
(a)



(b)

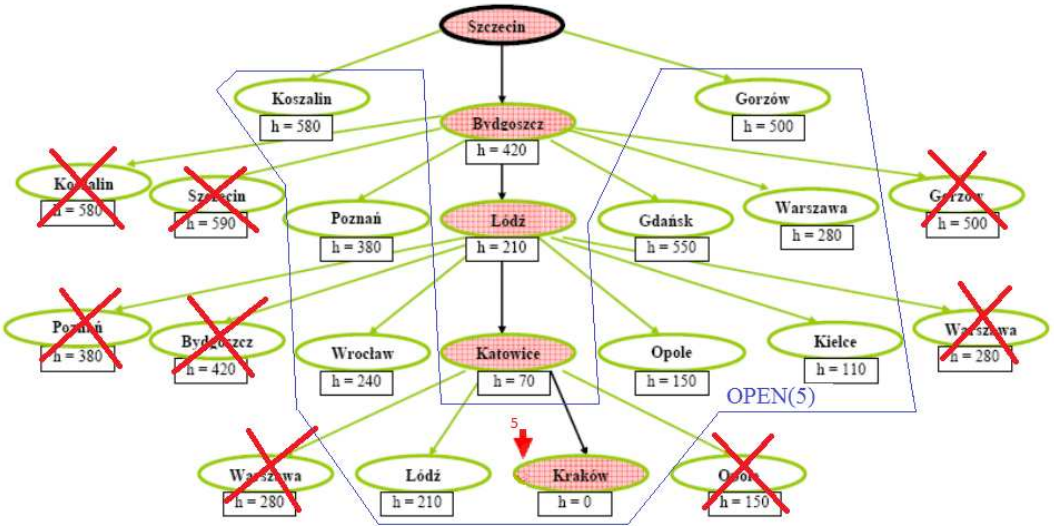


(c)

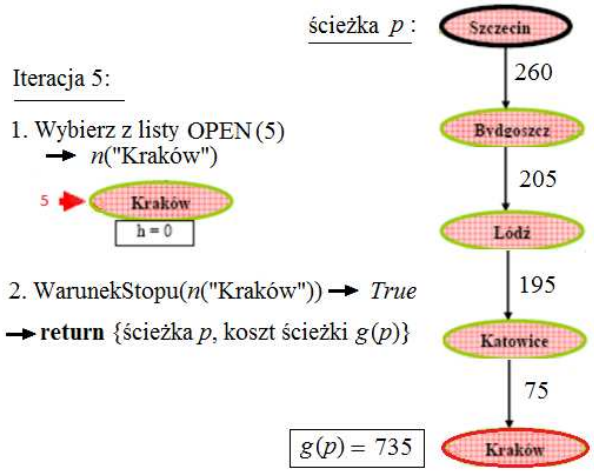


(d)

Rys. 7.3. Pierwsze cztery kroki przeszukiwania grafu z rys. 6.1 według poinformowanej strategii „zachłannej”: a) wybór węzła „Szczecin” (jedyny i najlepszy węzeł na liście OPEN(1)) i generowanie jego następników, b) wybór węzła „Bydgoszcz, najlepszego z węzłów na liście OPEN(2) – spośród jego następników usuwane są węzły „Koszalin, „Szczecin” i „Gorzów”, gdyż równoważne im węzły już istnieją w drzewie przeszukiwania (decyzyjnym); c) wybór węzła „Łódź”, najlepszego z węzłów na liście OPEN(3); d) wybór węzła „Katowice”, najlepszego z węzłów na liście OPEN(4).



(a)



(b)

Rys. 7.4. Ilustracja ostatniego kroku przeszukiwania zadanego grafu według strategii zachłannej: (a) wybór węzła „Kraków" jako najlepszego węzła na liście OPEN(5); (b) sprawdzenie warunku zatrzymania („stopu") dla węzła „Kraków" kończy się wynikiem pozytywnym („*True*") co skutkuje zakończeniem przeszukiwania i wynikiem w postaci ścieżki „p" w drzewie przeszukiwania o rzeczywistym koszcie, $g(p)= 735$ (liczonym jako suma kosztów akcji przejść pomiędzy miastami podanych w grafie problemu).

Zbadajmy cechy strategii „zachłannej":

1. zupełność? nie – może utknąć w pętli;
2. czas? $O(b^m)$, ale dobra heurystyka może dać znaczącą poprawę;
3. pamięć? $O(b^m)$, gdyż utrzymuje wszystkie wizytowane węzły w pamięci.
4. optymalność? nie (np. wybrano drogę przez *Bydgoszcz* o koszcie 735 zamiast drogi optymalnej przez *Gorzów* o koszcie 705).

Zasadniczą **wadą** strategii **zachłannej** jest brak gwarancji uzyskania optymalnego rozwiązania (w sensie minimalizacji rzeczywistego sumarycznego kosztu ścieżki rozwiązania). Tej wady nie posiada następna omawiana strategia przeszukiwania grafu: **A***.

7.3. Przeszukiwanie A*

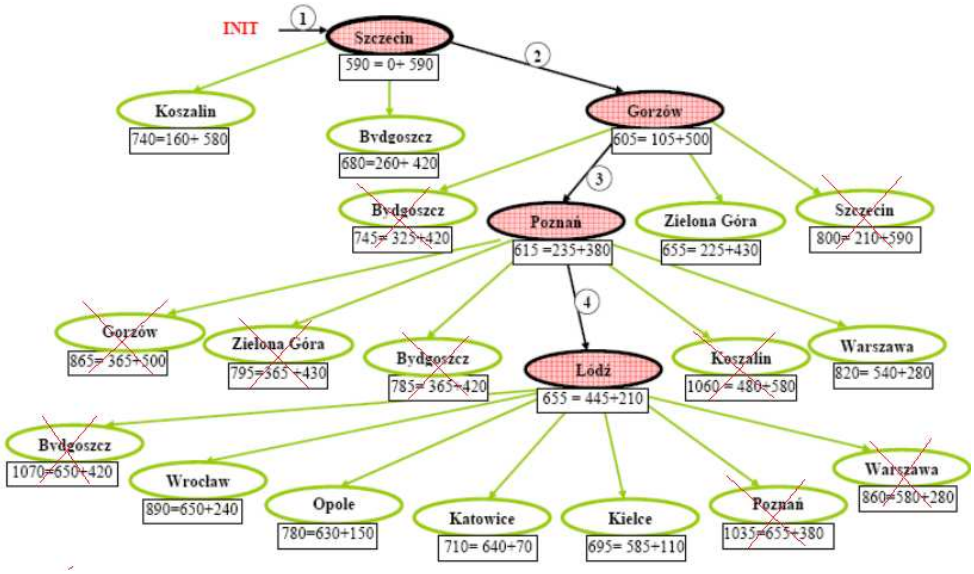
Idea strategii A* to: unikać rozwijania ścieżek, które już dotąd są kosztowne a poza tym niezbyt „obiecujące” pod względem możliwości szybkiego dotarcia do celu. Jej realizacja jest możliwa dzięki stosowaniu pełnej funkcji kosztu węzła:

$$f(n) = g(n) + h(n)$$

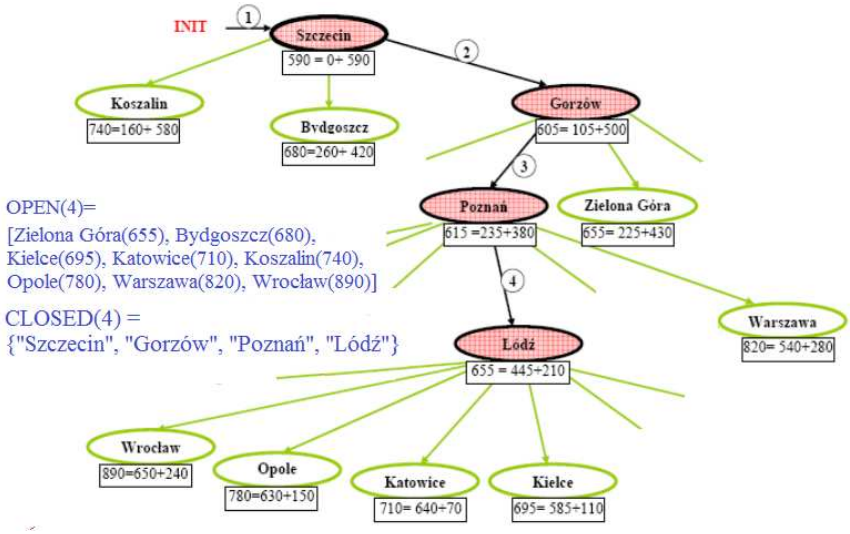
uwzględniającej zarówno koszt dotarcia do węzła n, (*składowa g(n)*), jak i przewidywany koszt przejścia z n do celu (składowa $h(n)$).

Dzięki temu, że $f(n)$ reprezentuje przewidywany **całkowity koszt** ścieżki prowadzącej od węzła startowego przez węzeł n do celu, a wizytowanie węzłów odbywa się w kolejności rosnącej wartości kosztu, to pod warunkiem **optymistycznego** oszacowania kosztów resztkowych, pierwsze napotkane rozwiązanie będzie jednocześnie najlepszym, czyli optymalnym.

Przykład. Ilustracja przeszukiwania grafu według strategii A* dla problemu „powrót ze Szczecina do Krakowa” (rys. 7.5, 7.6).

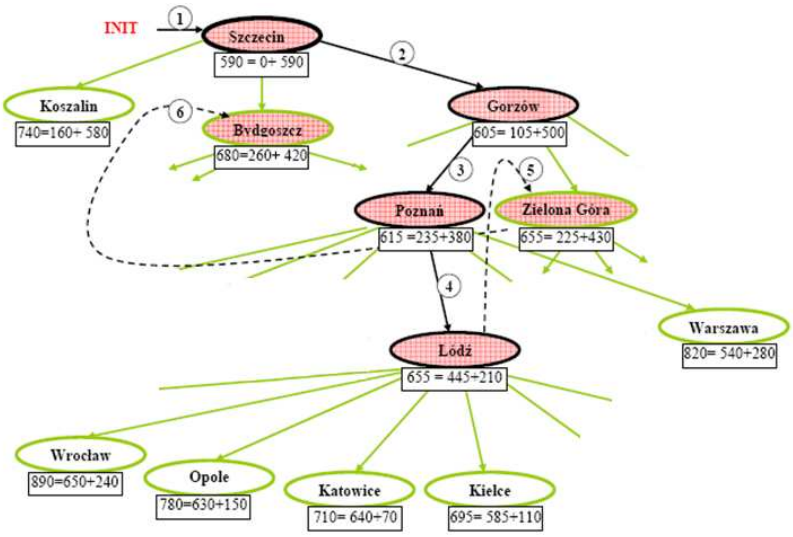


(a)

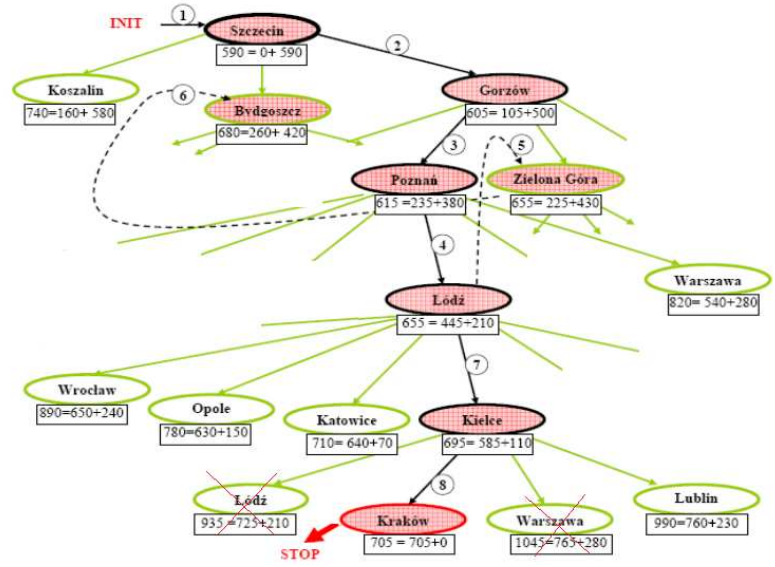


(b)

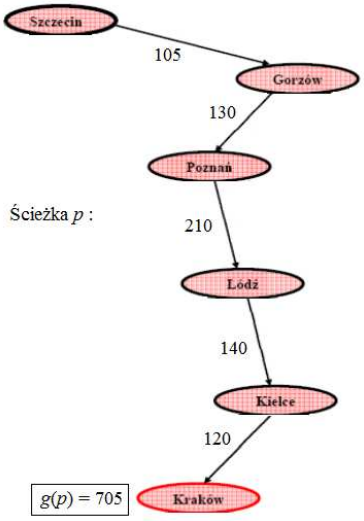
Rys. 7.5. Przeszukiwanie grafu dla zadanego problemu (rys. 7.1, tab. 7-1) według strategii A*: (a) drzewo przeszukiwania po 4 iteracjach wyboru i rozszerzania aktualnie najlepszego węzła (tzn. o najmniejszym koszcie *f*(*n*); jeśli generowany jest węzeł równoważny z węzłem, już istniejącym w drzewie przeszukiwania, to jest on dodawany do drzewa (listy OPEN) pod warunkiem posiadania niższego kosztu *g*(*n*) od węzła równoważnego w drzewie; (b) lista OPEN ma charakter globalny – podobnie jak dla strategii zachłannej.



(a)



(b)



(c)

Rys. 7.6. Ciąg dalszy procesu z rys. 7.5 - przeszukiwania grafu według strategii A*: a) drzewo przeszukiwania po 6 iteracjach; (b) drzewo przeszukiwania po 8 iteracjach – po wyborze węzła „Kraków” i sprawdzeniu warunku stopu cel został osiągnięty; (c) znaleziona ścieżka jest optymalna.

7.3.1. Implementacja strategii A*

Szczegółowy opis algorytmu implementującego strategię przeszukiwania A* podano w tabeli 7-3. Dzięki istnieniu zbioru CLOSED, do którego wstawiane są wizytowane węzły, i dzięki krokowi 6, uniemożliwiającemu wielokrotne wybieranie równoważnych węzłów (tego samego stanu problemu), algorytm realizuje przeszukiwanie **grafu**, a nie jedynie - drzewa.

Tab. 7-3. Algorytm A* dla przeszukiwania grafu.

1		INIT: Pobierz węzeł startowy s i umieść go w zbiorze OPEN. Ustaw $f(s)=0$, $g(s)=0$.
2		Pobierz z OPEN węzeł n o najmniejszej wartości funkcji $f(n)$ i umieść go w zbiorze CLOSED.
3		JEŚLI (n jest węzłem końcowym) TO zakończ i zwróć g(n) oraz całą ścieżkę od s do n.
4		Znajdź węzły następców n - niech będą nimi: $n_1' \ldots n_k'$.
5		Dla każdego z następców $n_1' \ldots n_k'$ oblicz koszt dojścia do niego: $g_i' = g(n) + c(n, ni')$.
6		Dla każdego z węzłów $n_1' \ldots n_k'$:
	a	JEŚLI (n_i' nie należy do zbioru OPEN ani do CLOSED) TO dodaj go do zbioru OPEN i ustaw: $g(n_i') = g_i'$, $f(n_i') = g_i' + h(n_i')$.
	b	JEŚLI (n_i' należy do zbioru OPEN lub CLOSED i $g(n_i') > g_i'$) TO ustaw $g(n_i') = g_i'$, $f(n_i') = g_i' + h(n_i')$, usuń ścieżkę od s do n_i'. Jeśli n_i' był w zbiorze CLOSED, to umieść go w zbiorze OPEN.
7		Powtórz od kroku 2.

Strategia A* posiada bardzo pożyteczne cechy:

- Zupełność? Tak, dla skończonej przestrzeni (nie istnieje nieskończenie wiele węzłów n, dla których, $f(n) \le f(G)$, gdzie G jest optymalnym celem).
- Czas? Potencjalnie wykładniczy ale znaczne zmniejszenie czasu przeszukiwania jest możliwe przy istnieniu dobrej heurystyki (bliskiej rzeczywistym kosztom).
- Pamięć? Wszystkie rozwijane węzły są pamiętane, z uwagi na możliwość wystąpienia cykli w grafie.
- Optymalność? Tak, jeśli heurystyka jest **dopuszczalna** to A* zawsze znajduje najlepsze rozwiązanie. Dalej wyjaśnimy, co oznacza dopuszczalna heurystyka - w skrócie mówiąc oznacza to, że oszacowanie musi być optymistyczne.

Przeszukiwanie A* dysponuje jeszcze jedną ciekawą cechą, jest nią **optymalna efektywność**: przy istnieniu *spójnej* heurystyki dla problemu, żadna inna strategia poinformowanego przeszukiwania nie rozwija mniej węzłów niż algorytm A* dla dotarcia do celu.

7.3.2. Dopuszczalna heurystyka

Heurystyka $h(n)$ jest dopuszczalna (ang. *admissible*), jeżeli dla każdego węzła n zachodzi

$$h(n) \le h^*(n),$$

gdzie $h^*(n)$ jest prawdziwym kosztem osiągnięcia celu z węzła n. Dopuszczalna heurystyka nigdy nie przecenia kosztu osiągnięcia celu, jest więc optymistycznym oszacowaniem rzeczywistego kosztu. Dla przykładu, heurystyka $h(n)$ podana w tab. 7-1 dla grupy problemów *„powrót do Krakowa"* nigdy nie przecenia faktycznej odległości liczonej jako suma odcinków drogi.

Możemy pokazać, że jeżeli $h(n)$ jest *dopuszczalna* heurystyką, to strategia przeszukiwania A* jest optymalną strategią przeszukiwania grafu. Dowód wynika z kilku obserwacji:

1. Strategia A* rozwija węzły w kolejności nie malejących wartości funkcji oceny (kosztu) *f* :

$$f_i \leq f_{i+1}$$

2. Strategia A* nie może wybrać węzła o określonym koszcie f_i zanim nie wybierze uprzednio wszystkich węzłów o niższym koszcie.
3. Jeśli heurystyka jest dopuszczalna to koszt każdego częściowego rozwiązania (ścieżki) nie przekracza rzeczywistego kosztu rozwiązania zawierającego tę ścieżkę.
4. Stąd pierwsze wybrane rozwiązanie końcowe nie może być gorsze od żadnego innego rozwiązania. Gwarantuje to osiągnięcie optymalnego rozwiązania.

7.3.3. Spójna heurystyka

Heurystyka jest spójna (ang. *consistent*), jeżeli jest dopuszczalna i dla każdego węzła n i dla każdego następnika n', wygenerowanego przez akcję a, spełniony jest „warunek trójkąta":

$$h(n) \leq c(n,a,n') + h(n'),$$

gdzie $c(n,a,n')$ oznacza koszt przejścia z n do n' za pomocą akcji a. O koszcie każdej akcji $c(n,a,n')$ z góry zakładamy, że jest nieujemny.

Spójność h oznacza, że:

1 $h(n) \leq h^*(n)$, składowa heurystyczna funkcji kosztu jest „dopuszczalna";

2) $f(n') = g(n') + h(n') = g(n) + c(n,a,n') + h(n') \geq g(n) + h(n) = f(n)$

Stąd, $f(n') \geq f(n)$, czyli funkcja kosztu $f(n)$ jest niemalejąca (monotoniczna) wzdłuż dowolnej ścieżki.

Można pokazać, że jeżeli heurystyka *h*(*n*) jest spójna, to strategia A* jest efektywnościowo optymalną strategią poinformowanego przeszukiwania grafu. Oznacza to, że żadna inna strategia optymalna, aby osiągnąć cel nie rozwija mniej węzłów niż strategia A*.

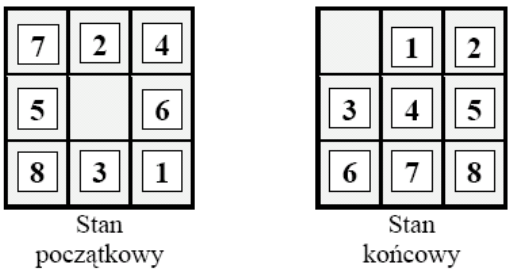
7.4. Wyznaczanie składowej heurystycznej

7.4.1. Dominująca heurystyka

Przykład. Wybrane składowe heurystyczne dla problemu „8-puzzli": $h_1(n)$ - liczba kafelków nie na swoim (docelowym) miejscu, $h_2(n)$ - całkowita odległość kafelków od swoich (docelowych) miejsc wyrażona w metryce Manhattan. Dla stanu początkowego na rys. 7.7, wartości obu heurystyk wynoszą:

h_1(stan początkowy) = 8;

h_2(stan początkowy) = 3+1+2+2+2+3+3+2 = 18



Rys. 7.7 Przykład problem “8-puzzli”

Heurystyki możemy ze sobą porównywać. Wprowadźmy określenie **dominującej** heurystyki.

Jeżeli $h_2(n) \geq h_1(n)$ dla każdego węzła n (i obie heurystyki są dopuszczalne) to h_2 dominuje nad h_1.

Wtedy h_2 jest bliższa rzeczywistym kosztom ale pozostaje optymistycznym oszacowaniem i dlatego też jest lepszą heurystyką niż h_1 dla poinformowanej strategii przeszukiwania.

7.4.2. Generowanie heurystyk metodą „złagodzonego problemu”

Zastanówmy się w jaki sposób „zautomatyzować” proces generacji dobrych heurystyk. Dla bardzo złożonych problemów człowiek jest w stanie podać analityczną postać (wzór matematyczny) jedynie dla obliczenia stosunkowo słabych heurystyk, np. dla heurystyki h_1 w problemie „N-puzzli”. Istotne jest też, aby mogła to zrobić maszyna dysponująca ograniczonym zestawem metod obliczeniowych. Stąd bierze się idea, aby wyznaczenie dobrej heurystyki potraktować jako oddzielny problem przeszukiwania, ale duży prostszy niż oryginalny problem.

Problem „uproszczony”, o zmniejszonych w porównaniu z oryginalnym problemem wymaganiach nakładanych na przejścia pomiędzy stanami, nazywamy problemem **złagodzonym**. W takiej sytuacji znalezienie optymalnego rozwiązania złagodzonego problemu (dla stanu początkowego n) stanowi jednocześnie dobrą, dopuszczalną heurystykę dla stanu n w oryginalnym problemie.

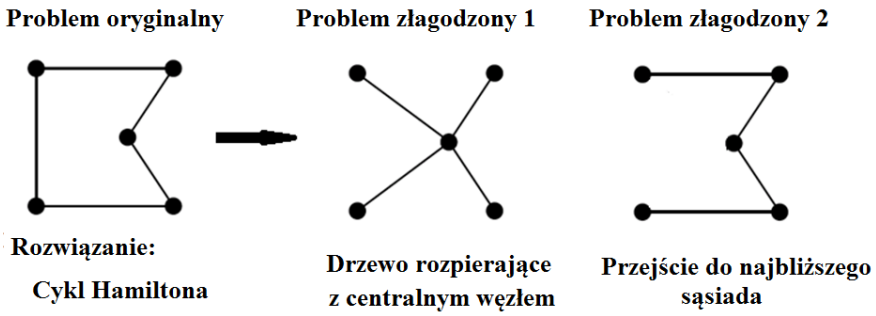
Np. jeśli złagodzimy problem „8-puzzli” tak, że kafelek może zostać przesunięty w dowolne miejsce, to heurystyka $h_1(n)$ dla oryginalnego problemu powstaje z rozwiązania problemu złagodzonego i wynosi tyle co koszt optymalnego rozwiązania rozpoczynanego w stanie n.

Podobnie, jeśli złagodzimy problem „8-puzzle” tak, że kafelek może zostać przesunięty w dowolną sąsiednią pozycję (tzn. nawet zajętą), to heurystyka $h_2(n)$ dla oryginalnego problemu odpowiada kosztowi optymalnej ścieżki w tym problemie złagodzonym.

Przykład.

Generowanie heurystyk metodą „złagodzonego problemu”. Dany jest „problem komiwojażera” – należy znaleźć najkrótszą trasę odwiedzenia n miast, odwiedzając każde miasto jedynie raz (jest to problemem o wysokiej złożoności $O(n!)$). Problem złagodzony wobec oryginalnego problemu – wyznaczyć drzewo rozpierające dla (pod-) zbioru węzłów pozostałych do odwiedzenia (jest to problem o złożoności jedynie $O(n^2)$) (rys. 7.8). Inny problem złagodzony może polegać na znalezieniu długości ścieżki wiodącej od aktualnego węzła przez węzły pozostałe jeszcze do odwiedzenia, których kolejność wynika z zasady wyboru najbliższego następcy.

Koszt znalezionego rozwiązania problemu złagodzonego o węźle początkowym n, stanowi wtedy optymistyczne oszacowanie kosztu resztkowego dla węzła n w problemie oryginalnym, gdy pozostaje jeszcze wizytowanie zadanego (pod-)zbioru miast (węzłów w grafie).



Rys. 7.8 Ilustracja problemu oryginalnego i „problemu złagodzonego” dla „problemu komiwojażera”: (a) oryginalny problem sprowadza się do znalezienia „cyklu Hamiltona” dla zadanego grafu problemu, o łukach etykietowanych kosztami przejść; (b) problem złagodzony 1 – dla zadanego węzła „centralnego” znaleźć drzewo rozpierające w grafie; (c) problem złagodzony 2 – lokalny wybór najlepszego następnika.

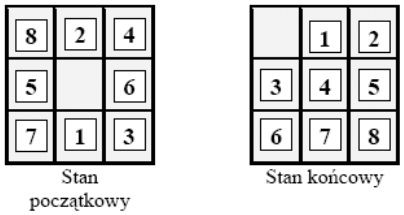
7.5. Pytania

1. Zdefiniować i zilustrować problem przeszukiwania.
2. Przedstawić strategie poinformowanego przeszukiwania? Kiedy strategia „najlepszy najpierw” jest poinformowana a kiedy nie?
3. Przedstawić strategie przeszukiwania: zachłanną i A*. Która z nich jest optymalna i w jakich warunkach?
4. Czy różni się przeszukiwania drzewa od przeszukiwania grafu?
5. Co oznaczają pojęcia: „dominacja heurystyki” i „problem złagodzony”?

7.6. Zadania

Zad. 7.1

W problemie „8 puzzli” znane są: stan początkowy i wymagany stan końcowy, jak na rysunku 7.9. Rozwiązać problem złagodzony wobec problemu 8 puzzli, w którym pojedyncza akcja polega na natychmiastowym przemieszczeniu (jedną akcją) jednego kafelka na wymagane miejsce docelowe (można kłaść jeden na drugi), stosując strategię **zachłanną przeszukiwania poinformowanego** przy zastosowaniu heurystyki odpowiadającej **metryce Manhattan**.



Rys. 7.9

Zad. 7.2

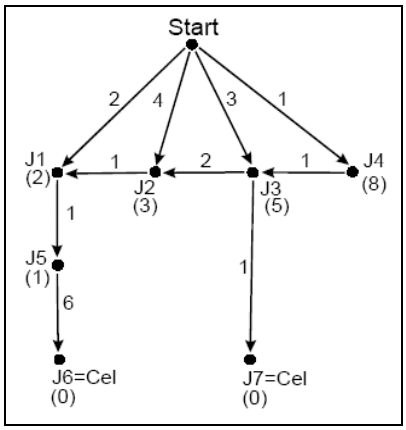
W podanym niżej grafie (rys. 7.10) pewnego problemu koszty operacji pomiędzy stanami podane są przez liczby przy łukach, a oszacowanie kosztów pozostałych jeszcze na drodze do celu (heurystyka) – przez liczby w nawiasach przy węzłach.

A) Zilustrować działanie strategii **przeszukiwanie A*** dla zadanego grafu problemu.

B) Czy heurystyka jest **dopuszczalna**?

C) Czy znaleziono **optymalne** rozwiązanie?

Uwaga: dwa stany spełniają warunek stopu (są docelowe), jednak tylko jeden z nich reprezentuje optymalne rozwiązanie.



Rys. 7.10: Graf problemu.

Zad. 7.3

Rozwiązać (zilustrowany na rys. 7.11) problem znalezienia optymalnej ścieżki (od węzła S do celu) stosując trzy poinformowane strategie przeszukiwania grafu:

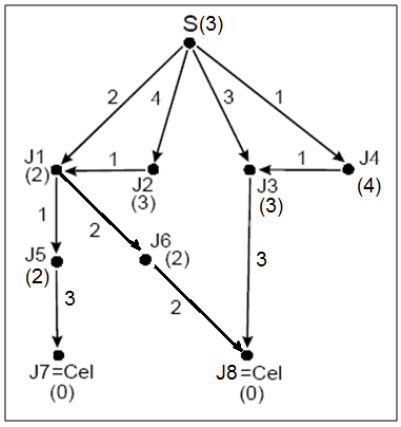
A) Strategię zachłanną,

B) Przeszukiwanie A*, i

C) Przeszukiwanie lokalne „przez wspinanie” (patrz punkt 8.2).

W nawiasach podano heurystykę (oszacowanie kosztów resztkowych) dla funkcji kosztu ścieżki.

W każdym z przypadków A), B) i C) podać: zasadę strategii, rozwijane drzewo decyzyjne i zwracaną ścieżkę. Porównać ze sobą wyniki i skomentować je.



Rys. 7.11

8. Losowe i lokalne przeszukiwanie

Rozpatrywane dotąd strategie przeszukiwania przestrzeni stanów poszukiwały globalnego optimum. Teraz zajmiemy się strategiami, które:

- wykorzystują elementy **losowości** lub
- mają charakter **lokalnego** przeszukiwania.

Wadą tych strategii będzie brak gwarancji uzyskania globalnie optymalnego rozwiązania, ale ich zaletą będzie znacznie mniejsza złożoność obliczeń (mniejsze wymagania na czas i pamięć) w porównaniu do strategii globalnego przeszukiwania.

Jeśli proces przeszukiwania jest „w pełni poinformowany" to z góry wiemy, jakie decyzje podejmować i nigdy nie zboczymy ze ścieżki optymalnego rozwiązania a liczba rozwijanych węzłów będzie liniową funkcją długości ścieżki – złożoność czasowa i pamięciowa takiej strategii wyniesie O(*m*), gdzie *m* jest długością ścieżki rozwiązania.

Niejako „na drugim biegunie" są strategie przeszukiwania, w których decyzje podejmowane są w losowy sposób lub w oparciu jedynie o lokalną obserwację. Siłą rzeczy te decyzje jedynie z pewnym prawdopodobieństwem (a nie pewnością) będą globalnie optymalne.

8.1. Losowość w przeszukiwaniu

8.1.1. Algorytmy losowego próbkowania

Najprostsza losowa strategia przeszukiwania to **losowe próbkowanie globalne** (tab. 8-1). Wraz z kolejnymi iteracjami algorytmu, zbiór odwiedzonych węzłów V będzie coraz większy, a prawdopodobieństwo, że pokryje się on ze zbiorem wszystkich węzłów grafu problemu, wzrasta do jedności.

Tab. 8-1: Algorytm losowego próbkowania przestrzeni stanów

```
function LosowePróbkowanie(problem, k)
returns stan końcowy
{ V = ∅ ;
    for i = 1 to k
    {       generuj losowo s_i ∈ Stany(problem);
            V = V ∪ {s_i}; //. zbiór CLOSED
            if (WarunekStopu(s_i)) return s_i;
    }
    return ∅;
}
```

W problemach, w których nie jesteśmy w stanie wygenerować a priori wszystkich stanów problemu, ale dla każdego węzła umiemy jedynie wygenerować jego najbliższych sąsiadów, znalezienie rozwiązania musi być wieloetapowe a wygenerowane następniki będą pochodzić z lokalnego sąsiedztwa aktualnego węzła. Czyli próbkowanie (losowy wybór następnika) ograniczy się do

aktualnego skraju drzewa przeszukiwania, reprezentującego wygenerowane uprzednio węzły, zaś skraj jest każdorazowo rozszerzany o lokalne następniki wybranego węzła (tab. 8-2).

Tab. 8-2 Strategia losowego wyboru następnika ze skraju drzewa przeszukiwania

```
function LosoweGenerowanieNastepnika(problem) return stan końcowy
{ k = 0; V_0 = s_0 = StanPoczatkowy(problem);
  A_0 = Nastepniki(s_0, problem) ; // zbiór OPEN
        repeat
        {       wybierz losowo s_{k+1} z A_k;
                V_{k+1} = V_k ∪ {s_{k+1}}; // zbiór CLOSED
                A_{k+1} = A_{k+1} ∪ Nastepniki(s_{k+1}, problem) – V_k;
                k = k+1;
        } until (s_k nie jest stanem końcowym, tzn.
                (WarunekStopu(problem, s_k) = True)
        return s_k;
}
```

8.1.2. Algorytm błądzenia przypadkowego

Metoda błądzenia przypadkowego charakteryzuje się l**okalnym** skrajem drzewa przeszukiwania. , Kolejne odwiedzane węzły skupiają się w swoich sąsiedztwach (tab. 8-3).

Tab. 8-3: Algorytm błądzenia przypadkowego

```
function BladziePrzypadkowe(problem, k) return stan końcowy
{ s_0 = StanPoczatkowy(problem);
 V = { s_0};
        for i = 1 to k
        {       generuj losowo s_i ∈ Nastepniki(s_{i-1});
                V = V ∪ {s_i};
                if (WarunekStopu(s_i)) return s_i;
        }
        return ∅;
}
```

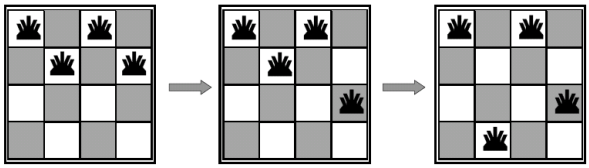
8.2. Przeszukiwanie lokalne poinformowane

W wielu problemach ścieżka prowadząca do celu sama w sobie nie jest taka ważna, jak sam fakt osiągnięcia celu. Np. w grach, przestrzeń stanów definiowana jest jako zbiór „pełnych" konfiguracji pionków. Celem gry (poszukiwanym rozwiązaniem) jest znalezienie konfiguracji spełniającej pew-

ne warunki. W takich sytuacjach możemy zastosować strategie przeszukiwania **lokalnego**, które zapamiętują jedynie pojedynczy stan „aktualny" i starają się go w sposób iteracyjny poprawiać.

W przeciwieństwie do dotychczas przez nas rozpatrywanych strategii globalnych, w których skraj drzewa obejmował wszystkie aktualne węzły niekońcowe, strategie lokalne operują na lokalnych skrajach, złożonych wyłącznie z następników aktualnie wybranego węzła. Dlatego też rozwiązanie znalezione w przeszukiwaniu lokalnym nie musi być globalnie optymalne.

Przykład. Problem *n*-królowych - ustawić *n* królowych na szachownicy o rozmiarze $n \times n$ tak, aby żadne dwie królowe nie znalazły się w tym samym wierszu, kolumnie i przekątnej (w zasięgu bicia) (rys. 8.1).



Rys. 8.1: Ilustracja stanów w problemie 4 królowych.

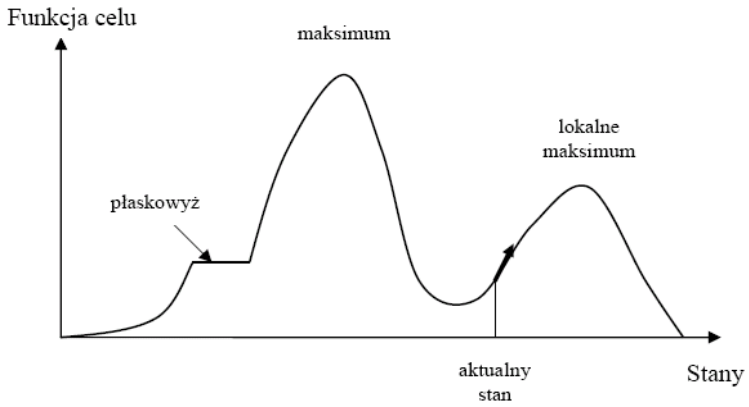
8.2.1. Przeszukiwanie przez „wspinanie"

Strategia lokalnego poinformowanego przeszukiwania określana jest poglądowo zdaniem: „*Wspinanie się na Mt. Everest w gęstej mgle będąc dotkniętym amnezją*" [4]. „Wspinanie się" oznacza wybór najlepiej ocenionego węzła następnika, „gęsta mgła" określa lokalny charakter skraju drzewa przeszukiwania, „amnezja" oznacza brak pamiętania przeszłych akcji i wcześniej odwiedzanych węzłów. Implementacja tej strategii nosi nazwę „przeszukiwanie przez wspinanie" (ang. *hill climbing*) i została zarysowana w tabeli 8-4.

Tab. 8-4. Algorytm „przeszukiwania przez wspinanie".

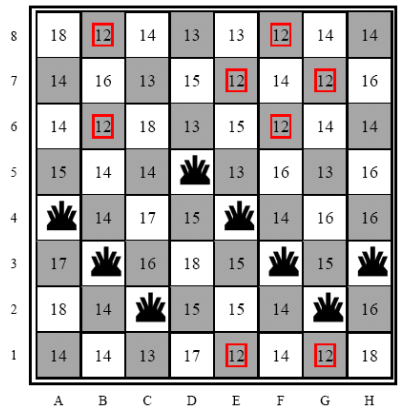
```
function HillClimbing (problem) return stan;
{
   węzełAktualny ← WĘZEŁ(STANPOCZĄTKOWY([problem]));
   while (True)
   {
         sąsiad ← NAJLEPSZYNASTĘPCA(węzełAktualny);
         if (OCENA(sąsiad) ≤ OCENA(węzełAktualny)
         then    return STAN(węzełAktualny);
         węzełAktualny ←sąsiad;
   }
}
```

Na rys. 8.2 zilustrowano zasadniczą wadę każdej strategii lokalnej - możemy utknąć w lokalnym optimum (zamiast minimalizacji kosztu zastosowano tu dualnie maksymalizację funkcji celu).



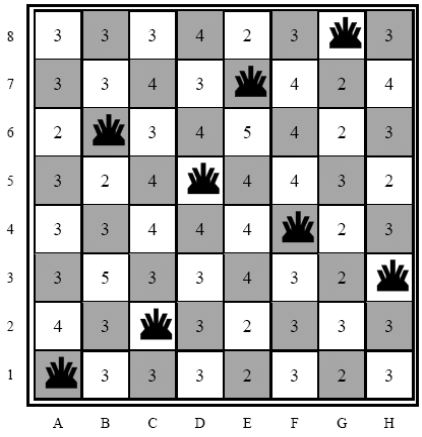
Rys. 8.2: Ilustracja lokalnego optimum znalezionego w przeszukiwaniu lokalnym.

Przykład. Przeszukiwanie przez „wspinanie” w zastosowaniu do problemu 8-królowych. Niech h oznacza liczbę atakujących się wzajemnie par hetmanów (bezpośrednio lub pośrednio).



Rys. 8.3: Przeszukiwanie lokalne zastosowane w problemie „8 królowych”.

Ponieważ stan początkowy (rys. 8.3) zawiera po jednej królowej w każdej kolumnie, więc uprościmy problem ograniczając ruch każdej królowej jedynie do swojej kolumny. Ocena stanu początkowego wynosi: $h = 17$. Liczba w każdym wolnym polu podaje ocenę h dla stanu powstałego ze stanu początkowego po przesunięciu do niego królowej znajdującej się w danej kolumnie. Czerwoną ramką zaznaczone zostały najlepsze oceny następników stanu początkowego, $h = 12$. Wybierana jest akcja przesunięcia królowej na najlepszą pozycję, czyli jedną z tych o ocenie 12. Iteracyjnie powtarzamy ten cykl (generowanie następników, ich ocena, wybór najlepszego) dopóki możliwa jest poprawa funkcji oceny. Jak wiemy, strategia lokalna nie gwarantuje osiągnięcia globalnego optimum. W tym przykładzie końcowy wynik, odpowiadający lokalnemu minimum funkcji oceny, jakie można osiągnąć z zadanego stanu początkowego to $h = 1$, podczas gdy idealne rozwiązanie posiada ocenę h=0. Na rys. 8.4 podano stan końcowy osiągany w tym przykładzie. Nie ma już możliwości poprawy oceny, gdyż wszystkie stany następne mają gorszą ocenę od aktualnego.

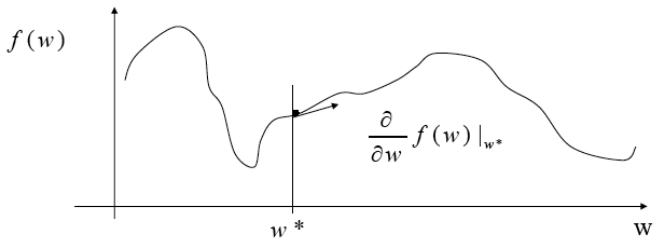


Rys. 8.4: Wynik przeszukiwania lokalnego dla stanu początkowego z rys. 8.3.

8.2.2. Przeszukiwanie lokalne w dziedzinie ciągłej

Przeszukiwanie z dodatnim gradientem (ang. „*gradient ascent search*") jest odpowiednikiem „przeszukiwania przez wspinanie" w dziedzinie ciągłych wartości (w ogólności: wektora) parametrów $\boldsymbol{w}$ **funkcji celu,** $y = f(x \mid \boldsymbol{w})$ (rys. 8.5). Jego zasada polega na iteracyjnej modyfikacji wektora $\boldsymbol{w}$:

$$\boldsymbol{w}_{i+1} = \boldsymbol{w}_i + \alpha \frac{\partial}{\partial \boldsymbol{w}} f(\boldsymbol{w})|_{w_i}$$



Rys. 8.5: Ilustracja przeszukiwania wzdłuż gradientu funkcji celu

Przeszukiwanie z ujemnym gradientem (ang. *gradient descent search*) to dualny problem poszukiwania lokalnego minimum funkcji celu:

$$\boldsymbol{w}_{i+1} = \boldsymbol{w}_i - \alpha \frac{\partial}{\partial \boldsymbol{w}} f(\boldsymbol{w})|_{w_i}$$

8.3. Symulowane wyżarzanie

Zasygnalizujemy tu inną grupę strategii rozwiązywania problemów, które zawierają niedeterministyczne (stochastyczne) elementy. Jedną z nich jest strategia „symulowanego wyżarzania". Można ją podsumować jako próbę poprawienia wad lokalnego poinformowanego przeszukiwania poprzez wysoce prawdopodobne wydostanie się z lokalnego optimum przez początkowo częste wykonywanie przypadkowych „złych" akcji, stopniowe zmniejszane wraz z obniżaniem sie parametru „temperatury".

Strategia „symulowanego wyżarzania" realizuje niedeterministyczne przejścia pomiędzy stanami. Jeśli losowe przejście poprawia sytuację to jest na pewno wykonywane, w przeciwnym razie wykonywane jest z pewnym prawdopodobieństwem mniejszym niż 1, zależnym od aktualnej wartości

parametru *T* (globalnej „temperatury”). Wartość ta stale maleje, co czyni takie przejścia coraz mniej prawdopodobnymi wraz z upływem czasu przeszukiwania. Można pokazać, że jeśli *T* zmniejsza się wystarczająco wolno to „symulowane wyżarzanie” znajduje globalne optimum z prawdopodobieństwem bliskim 1. Implementacja strategii „symulowanego wyżarzania” została zarysowana w tab. 8-5. Zachowano tradycyjne określenie „Energia” dla oceny stanu.

Tab. 8-5. Algorytm „symulowanego wyżarzania”.

```
function SymulowaneWyżarzanie( problem, temp[] ) return stan będący rozwiązaniem;
{
  current ←WĘZEŁ(PoczątkowyStan[problem]);
  for t←1 to rozmiar(temp) do
  {
     T← temp[t];   // T jest coraz mniejsze
     if (T == 0) then return current;          // Koniec przeszukiwania
     next ←losowo wybrany następca dla current;
     dE = ENERGIA[next] – ENERGIA[current]; // Energia == Ocena
     if (dE >0 ) then current ←next;
     else current ←next  tylko z prawdopodobieństwem exp(dE/T);
  }
  return current;
}
```

8.4. Algorytmy genetyczne

Przeszukiwanie wiązki (ang. „*beam search*”) polega na jednoczesnym rozwijaniu pewnej liczby ścieżek. Zasada:

- Jednoczesny wybór *k* stanów zamiast jednego; wybór *k* najlepszych następników.
- Nie jest to tożsame z równoległymi *k* pojedynczymi przeszukiwaniami, gdyż rozwijane są tu jedynie „dobre” stany.

Z uwagi na problem, że często wszystkie *k* stanów prowadzi do tego samego lokalnego optimum, pojawia się pomysł losowego wyboru następników.

Idea:

- wybierać losowo *k* następników, z tendencją do wyboru „dobrych” następników.

Zauważmy pewną analogię z naturalną selekcją:

- następcy są podobni do swoich rodziców, zdrowsi osobnicy z większą pewnością mają dzieci, czasami zdarzają się przypadkowe mutacje.

Algorytm genetyczny (tab. 8-6) stanowi połączenie stochastycznego lokalnego przeszukiwania wiązki i metody generowania następników z połączenia par stanów:

1. Pojedyncze rozwiązanie (stan) jest postaci sekwencji „genów”.
2. Wybór następników ma charakter losowy, z prawdopodobieństwem proporcjonalnym do ich oceny (jakości, ang. *fittness*).

3. Stany wybrane do reprodukcji są parowane w sposób losowy, niektóre geny są mieszane a niektóre mutują.

Tab. 8-6: Funkcja „algorytm genetyczny.

funkcja GA (próg-oceny, p, r, m) **zwraca** stan

{ **P**← { p losowo wybranych stanów }

FOR każdy stan h w **P**: OBLICZ-OCENĘ (h)

WHILE [$\mathbf{max}_h$ OCENA(h)] < PRÓG-OCENY DO

{ 1. Losowy wybór : dodaj (1-r)· p stanów do $\mathbf{P}_s$

$Pr(h_i)$= OCENA(h_i)/ SUMAOCEN (wszystkich stanów z $\mathbf{P}_s$)

2. Krzyżowanie: losowo wybierz r·p/2 par stanów z $\mathbf{P}_s$.

Dla każdej pary (h_j, h_k), zastosuj OPERATOR KRZYŻOWANIA.

Dodaj wyniki do $\mathbf{P}_s$

3. MUTACJA: inwersja losowo wybranego bitu w m· p losowo wybranych stanach z $\mathbf{P}_s$

4. Podstaw: **P**← $\mathbf{P}_s$

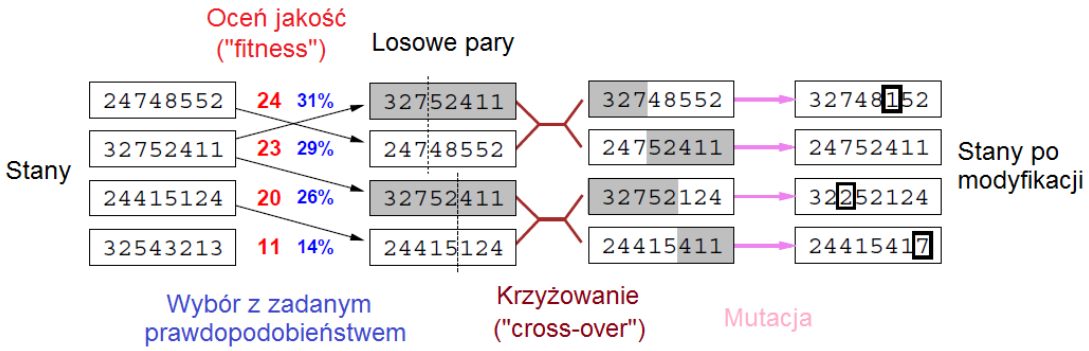
5. FOR każdy stan h w **P**: OBLICZ-OCENĘ (h)

}

return stan z **P** o najwyższej ocenie

}

Przykład



Rys. 8.6: Ilustracja algorytmu genetycznego

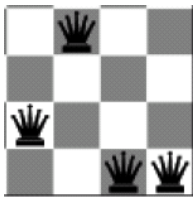
8.5. Pytania

1. Przedstawić losowe, niepoinformowane algorytmy przeszukiwania („próbkowanie” globalne i lokalne, „błądzenie przypadkowe”).
2. Przedstawić poinformowane przeszukiwanie lokalne („przez wspinanie”).
3. Przedstawić poinformowane losowe przeszukiwanie („symulowane wyżarzanie”).
4. Omówić algorytm genetyczny.

8.6. Zadania

Zad. 8.1

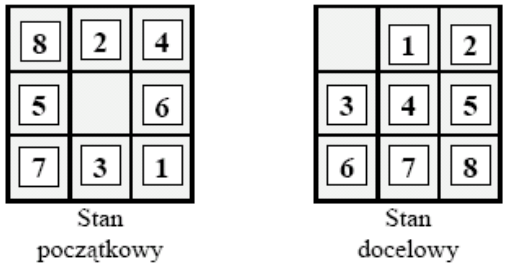
Niech w problemie 4 królowych oszacowanie kosztów resztkowych (heurystyka) wyrażone jest liczbą zagrażających sobie wzajemnie par (tzn. dwie królowe są w tym samym wierszu lub kolumnie lub przekątnej). Przyjmijmy, że możliwe do wykonania akcje polegają na przesuwaniu pionka w kolumnie (w każdej kolumnie jest jeden pionek). Dla podanego poniżej stanu początkowego wykonać symulacje działania algorytmu lokalnego przeszukiwania „przez wspinanie" (rys. 8.7).



Rys. 8.7: Stan początkowy.

Zad. 8.2

Dla poniższego przykładu - problemu „8 puzzli" – podano przykładowy **stan początkowy** i wymagany **stan docelowy** (rys. 8.8).



Rys. 8.8

A) Rozwiązać **problem złagodzony** wobec problemu 8 puzzli, w którym pojedyncza akcja polega na natychmiastowym przemieszczeniu jednego kafelka na wymagane miejsce docelowe (można kłaść jeden na drugi), stosując algorytm **przeszukiwania lokalnego „przez wspinanie"** przy zastosowaniu funkcji **heurystycznej** odpowiadającej metryce Manhattan.

B) Podać: wykonywane **akcje** (uzasadnić dlaczego) i rozwijane **drzewo decyzyjne**.

9. Gry dwuosobowe

W nietrywialnych grach nie da się rozwiązać zagadnienia wyboru następnego ruchu ekstensywną metodą rozważenia wszystkich możliwych sekwencji ruchów własnych i przeciwnika. Oznacza to, że wybór ruchu w grach dwuosobowych, tak jak rozwiązywanie innych złożonych zadań, wymaga odpowiednich strategii przeszukiwania.

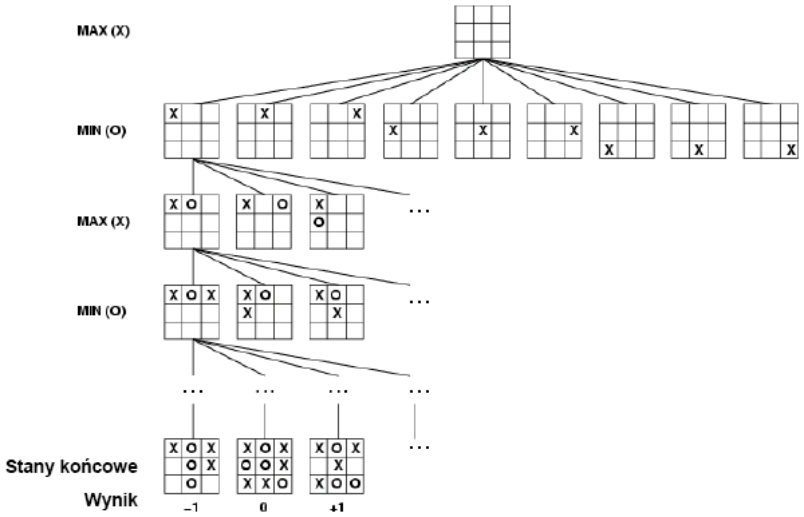
Ponieważ przeciwnik jest „nieprzewidywalny", należy określić własny ruch przewidując „najgorszą dla nas" decyzję przeciwnika (→ strategia „Mini-max")

Ze względu na ograniczenie czasu gry znalezienie optymalnego ruchu metodą przejrzenia wszystkich możliwych sekwencji ruchów prowadzących do celu może nie być możliwe. Wtedy trzeba aproksymować najlepsze rozwiązanie (→ „obcięty Mini-max").

9.1. Drzewo gry typu „Mini-max"

Ograniczmy nasze rozważania do gier dwuosobowych o przeciwstawnych celach) o ruchach wykonywanych na przemian przez obu graczy. Rozważamy model gry w postaci przeszukiwania specyficznego drzewa typu „Mini-max" (rys. 9.1):

- Występują 2 rodzaje węzłów: Min i Max,
  - węzeł Min reprezentuje stan gry, w którym ruch należy do przeciwnika,
  - węzeł Max odpowiada decyzji naszego gracza.
- Ocena stanu gry propagowana jest „od dołu na górę" (od liści do korzenia drzewa) - ocenę węzła rodzica typu Max ustawiamy na wartość maksymalną spośród jego węzłów potomnych (następników), ocena węzła rodzica typu Min jest minimalną spośród jego następników;
- Możliwe akcje:
  - „nasz" gracz wybiera akcję odpowiadającą przejściu do najlepszego spośród jego następców;
  - „przeciwnik" wykonuje ruch odpowiadający przejściu do najgorszego następnika.

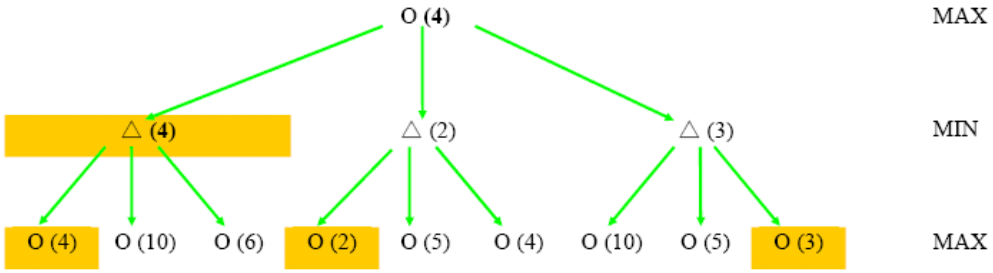


Rys. 9.1: Ilustracja drzewa Mini-max

Ocena węzła odpowiada możliwemu do osiągnięcia wynikowi gry, reprezentowanemu przez liść drzewa przeszukiwania (rys. 9.2). Np. możliwy wynik w grze „kółko i krzyżyk”:

+1 = wygrana, 0 = remis, -1= porażka.

Niektóre gry mogą jednak kończyć się z różnymi wynikami punktowymi. W ogólności funkcja Wynik(*stan*) dostarcza oceny liczbowej każdego stanu końcowego gry.



Rys. 9.2: Ilustracja wyznaczenia wyniku gry w liściach drzewa i jego propagacji w górę drzewa do korzenia.

9.2. Strategia Mini-max

„Mini-max” zakłada **idealną** rozgrywkę dla **deterministycznych** 2-osobowych gier naprzemiennych. Strategia polega na wybieraniu przez „naszego gracza” ruchu do pozycji z najwyższą wartością i na przyjęciu założenia, że przeciwnik wybiera dla nas najgorszy ruch.

Tym samym strategia „Mini-max” realizuje cel: *uzyskać najlepszy możliwy wynik w grze z najlepszym przeciwnikiem*.

Implementacja strategii „Mini-max” składa się z funkcji **MiniMaks()** (Tab. 9-1) i jej podfunkcji **MaksOcena()** (Tab. 9-2) i **MinOcena()** (Tab. 9-3).

Tab. 9-1 Funkcja **MiniMaks**.

```
function MiniMaks(stan) : returns akcja
{
    v := MaksOcena(stan);
    wybierz akcja = (stan, stan_N),
                    gdzie: stan_N ∈ Następne(stan), Wynik(stan_N) == v ;
    return akcja;
}
```

Tab. 9-2 Funkcja **MaksOcena**.

```
function MaksOcena(stan) : returns najlepszy_możliwy_wynik
{
    if (TerminalTest(stan)) then return Wynik(stan);
    v := - ∞ ;
    for (s ∈ Następne(stan)) do v := max(v, MinOcena(s));
    return v;
}
```

Tab. 9-3 Funkcja **MinOcena**

```
function MinOcena(stan) : returns najgorszy_możliwy_wynik
{
        if (TerminalTest(stan)) then return Wynik(stan);
        v := ∞ ;
        for (s ∈ Następne(stan)) do v := min(v, MaksOcena(s));
        return v;
}
```

Własności przeszukiwania Mini-Max:

- Zupełność? Tak (jeżeli drzewo jest skończone)
- Optymalność? Tak (jeżeli przeciwnik jest racjonalny)
- Czas? $O(b^m)$ (przeszukujemy całe drzewo)
- Pamięć? O(bm) (stosujemy przeszukiwanie w głąb)

  gdzie: b – średni stopień rozgałęzienia drzewa, m – długość ścieżki rozwiązania.

Dla szachów szacujemy: b≈35, m≈100, dla "*rozsądnych*" rozgrywek, tzn. wtedy gdy spotykają się gracze o podobnych wysokich umiejętnościach. W praktyce, przy tak dużych wartościach parametrów, dokładne rozwiązanie problemu strategią **Mini-maks** jest niemożliwe.

Wymagane jest usprawnienie tej strategii. Taką formą jest strategia przeszukiwania drzewa Min-Max zwana „cięciami alfa-beta".

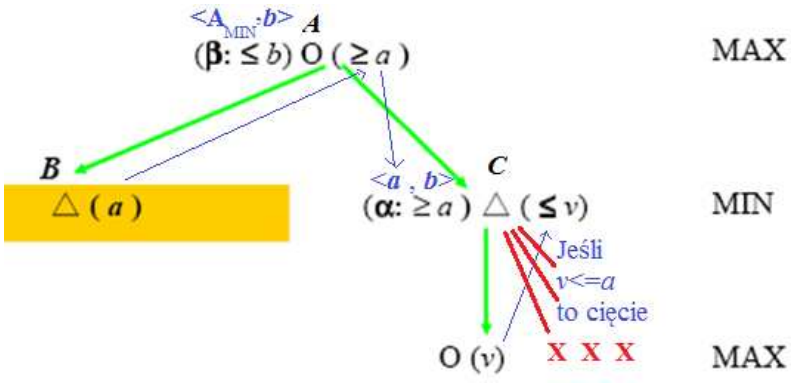
9.3. Cięcia alfa-beta

9.3.1. Zasada przycinania drzewa

Niech *a* jest oceną najlepszego wyniku w dotychczas przeanalizowanym poddrzewie *B* dla korzenia *A* typu MAX.

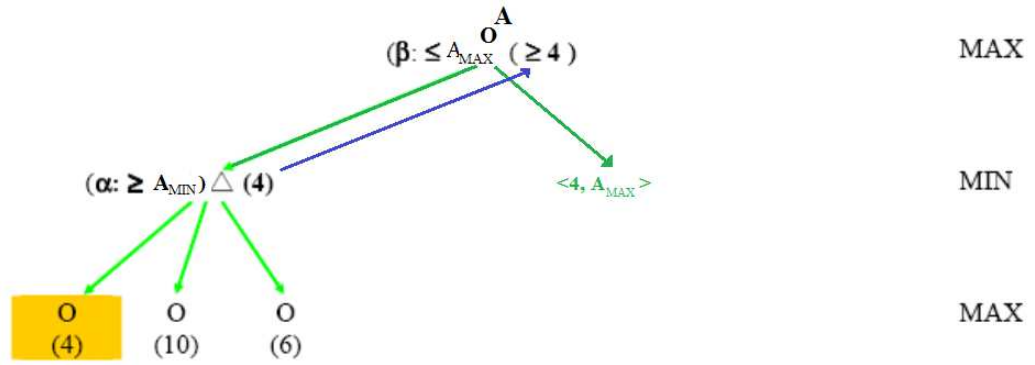
Jeśli podczas analizy kolejnych gałęzi *C* węzła *A* jego ocena *v* jest gorsza niż *a*, to bez szkody dla optymalnego rozwiązania zrezygnujemy z *v* , czyli przycinamy gałąź *C* (cięcie „alfa").

Podobnie zrobimy dla gałęzi węzła *B* typu MIN wtedy, gdy stwierdzimy, że gałąź dla *B* będzie miała ocenę *w* i zachodzi *w*≥*b* : możemy przyciąć tę gałąź (cięcie „beta") (rys. 9.3).

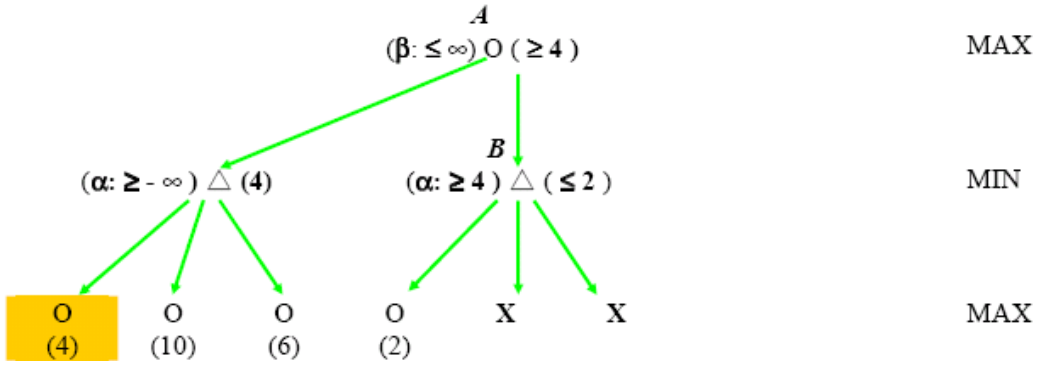


Rys. 9.3

Ustawienie warunku przycinania dla gałęzi węzła *A* typu MAX (rys. 9.4(a)). Dotychczas przeanalizowana gałąź dostarcza ocenę 4 (zakładamy, że 4 > A_{MIN}). Teraz interesują nas już wyłącznie te gałęzie *A*, które mogą dać wyższą ocenę niż 4. Stąd dalsze gałęzie węzła *A* będą posiadały ograniczenie: $\alpha = 4$.



(a)

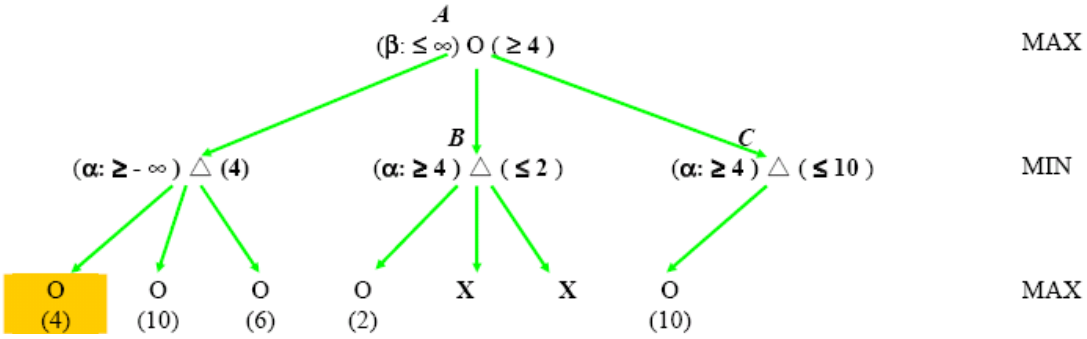


(b)

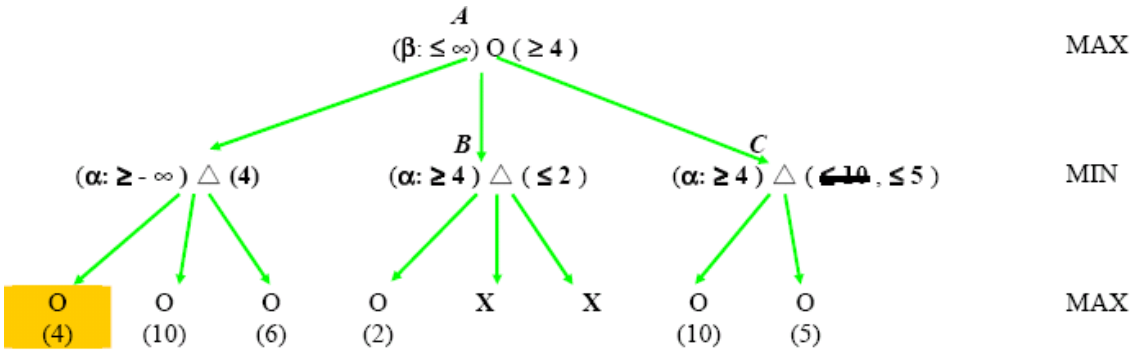
Rys. 9.4

Realizacja „cięcia alfa" (rys. 9.4(b)). Dla węzła *B* typu MIN dotychczas przeanalizowana gałąź dostarcza ocenę 2. W kontekście oceny 4 dla węzła nadrzędnego *A,* warunek ((α=4) > 2) „blokuje" rozpatrywanie pozostałych gałęzi węzła *B* i przycina *B*.

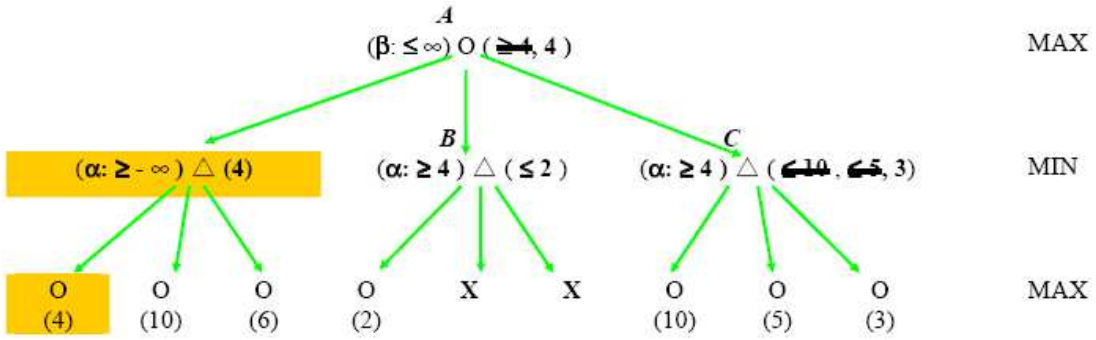
Ustawienie warunku przycinania dla gałęzi węzła MIN (rys. 9.5(a). Dla węzła *C* typu MIN jej pierwsza przeanalizowana gałąź dostarcza ocenę 10. Nie możemy wykonać „cięcia alfa" pozostałych gałęzi *C*, gdyż (10 > α) nadal jest spełniony, ale ustawiamy ograniczenie (β = 10) dla potomków węzła *C*.



(a)



(b)



(c)

Rys. 9.5

Dla węzła *C* druga gałąź daje ograniczenie górne 5. Nadal nie możemy wykonać przycinania pozostałej gałęzi *C,* gdyż warunek α nadal jest spełniony w węźle *C*, jednak ponownie ustawiamy nowe ograniczenie górne, (β = 5), dla potomków węzła *C* (rys. 9.5(b)).

Ostatnia gałąź węzła C dała wynik 3. W węźle następuje maksymalizacja wyników jej gałęzi: max(4, 2, 3) = 4. Wybierany jest ruch prowadzący do węzła typu MIN o ocenie 4 (rys. 9.5(c)). Pomimo wykonanego cięcia wynik jest ten sam co dla podstawowego algorytmu Mini-Max.

Podsumowując powyższy przykład możemy podać ogólne zasady cięć alfa-beta.

Cięcie alfa: oceniając węzeł MIN przez minimalizację ocen węzłów potomnych typu MAX możemy zakończyć wyznaczanie ocen węzłów potomnych natychmiast po stwierdzeniu, że ocena węzła MIN nie będzie wyższa niż jej ograniczenie dolne α (pochodzące z nadrzędnego węzła MAX).

Cięcie beta: oceniając węzeł MAX przez maksymalizację ocen węzłów potomnych typu MIN możemy zakończyć wyznaczanie ocen węzłów potomnych natychmiast po stwierdzeniu, że ocena węzła MAX nie będzie niższa niż jej ograniczenie górne β (pochodzące z nadrzędnego węzła MIN).

9.3.2. Implementacja strategii „Mini-Max z cięciami alfa-beta”

Niech A_{MIN} oznacza najniższą możliwą ocenę w danej grze;

Podobnie A_{MAX} niech oznacza najwyższy możliwy wynik.

Jeśli nie można ustalić tych wartości to przyjmujemy: $A_{MIN} = -\infty$; $A_{MAX} = \infty$.

Funkcja **CięciaAlfaBeta()** i jej podfunkcje **MaksOcena()** i **MinOcena()** podane są w tabeli 9-4.

Tab. 9-4 Implementacja przeszukiwania „cięcia alfa-beta”.

```
function CięciaAlfaBeta(stan, A_MIN, A_MAX) : returns akcja {
        v := MaksOcena(stan, A_MIN, A_MAX);
        wybierz akcja = (stan, stan_N),
                        gdzie: stan_N ∈ Następne(stan), Wynik(stan_N) == v ;
        return akcja;
}
function MaksOcena(stan, α, β) : returns najlepszy_wynik {
        if (TerminalTest(stan)) then return Wynik(stan);
        v := α ;
        for (s ∈ Następne(stan)) do
        begin  v := max(v, MinOcena(s, α, β ));
                if v ≥ β then return v;  // Cięcie beta
                α := max(α, v);
        end;
        return v;
}
```

Dodatkowe parametry:

α - największa wartość węzła Max dla ścieżki rozwiązania

β - najmniejsza wartość węzła Min dla ścieżki rozwiązania

```
function MinOcena(stan, α, β) : returns najgorszy_wynik  {
        if (TerminalTest(stan)) then return Wynik(stan);
        v := β ;
        for (s ∈ Następne(stan)) do
        begin  v := min(v, MaksOcena(s, α, β ));
                if v ≤ α then return v; // Cięcie alfa
                β := min(β , v);
        end;
        return v;
}
```

9.3.3. Własności strategii „cięć alfa-beta”

Przycinanie nie ma wpływu na końcowy rezultat przeszukiwania. Osiągamy **ten sam wynik**, który dałoby zastosowanie podstawowej strategii Mini-Max.

Sprzyjające uporządkowanie ruchów (następników węzła) poprawia efektywność przycinania. Przy "perfekcyjnym uporządkowaniu” następników złożoność czasowa algorytmu wynosi $O(b^{m/2})$. Dzięki temu możliwe staje się rozwiązanie za pomocą „Mini-max-u z cięciami α-β” problemów o dwa razy większej głębokości drzewa przeszukiwania niż w przypadku klasycznej strategii Mini-Max.

9.4. Heurystyczna ocena – „obcięty Mini-Max”

W praktyce występują ograniczenia zasobów. Załóżmy, że mamy 100 sekund czasu na wykonanie ruchu i możemy przeglądać węzły z prędkością 10^6 węzłów na sekundę → czyli mamy czas na przeglądanie do 10^8 węzłów dla wykonania jednego ruchu.

9.4.1. Zasada modyfikacji „Mini-Max”-u w praktyce

1. Wykonujemy „test odcięcia” dla ścieżki (tu: funkcja C*utoff*). Np., test polega na ograniczeniu głębokości drzewa do wartości zadanej parametrem D.
2. Wprowadzamy funkcję oszacowania stanu (heurystyka) (tu: Ocena()) – dokonuje ona szacunkowej oceny (niekońcowej) konfiguracji gry, reprezentowanej przez aktualny węzeł drzewa.

Przykład funkcji oceny stanu.

- Dla warcabów, zazwyczaj stosujemy liniową sumę ważoną cech pionków w danej pozycji:

$$Ocena(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$$

- Np.: $w_1 = 9$ (szacunkowa moc damki),

$$f_1(s) = \text{(liczba białych damek)} - \text{(liczba czarnych damek)}$$

  itp.

- W ocenie uwzględnia się również wagi poszczególnych pozycji na planszy i wagi dla wzajemnych położeń pionków obu graczy.

9.4.2. Implementacja obciętego „Mini-Max”-u

Implementacja strategii „obciętego Mini-Max”-u jest prawie identyczna z podstawowym algorytmem „Mini-Max” (tab. 9-5):

1. Po sprawdzeniu warunku stopu – TerminalTest() - mamy teraz dodatkowo sprawdzanie warunku obcinania ścieżki – funkcją Cutoff() - i ewentualne obcięcie z podaniem wartości heurystycznej oceny aktualnego stanu.
2. Zamiast funkcji Wynik(), podającej *rzeczywistą ocenę końcowego stanu gry* dla pierwszego gracza, w sytuacji obcinania wywołujemy *funkcję oszacowania* stanu: Ocena().

Czy ta strategia dobrze działa w praktyce? Odpowiedzmy na to na przykładzie. Dla szachów, niech liczba możliwych stanów (na początku gry), które mogą być wizytowane w zadanym czasie wynosi: $b^d = 10^8$. Przy typowym stopniu rozgałęzienia początkowego drzewa dla gry w szachy, b = 35, odcinanie musiałoby w takiej sytuacji nastąpić po, d = 5, posunięciach. Przewidywanie przez gracza jedynie 5 ruchów do przodu jest w praktyce oznaką kiepskiego gracza w szachy! Siła gry gracza w szachy oceniana jest bowiem według następującej skali:

- przewiduje 4 ruchy ≈ początkujący gracz,
- przewiduje 8 ruchów ≈ program szachowy na typowym komputerze PC, mistrz szachowy,

Tymczasem program na komputerze *Deep Blue* lub arcymistrz szachowy potrafią przewidywać 12 ruchów do przodu.

Tab. 9.5 Implementacja strategii „obcięty Mini-Max”

```
function ObciętyMiniMaks(stan, D) : returns akcja {
        v := MaksOcena(stan, D);
        wybierz akcja = (stan, stan_N),
                        gdzie: stan_N ∈Następne(stan), Wynik(stan_N) == v ;
        return akcja;
}
function MaksOcena(stan, d) : returns najlepszą_ocena
{
        if (TerminalTest(stan)) then return Wynik(stan);
        if (Cutoff(stan, d)) then return Ocena(stan);
        v := - ∞ ;
        for (s ∈ Następne(stan)) do v := max(v, MinOcena(s, d-1));
        return v;
}
function MinOcena(stan, d) : returns najgorsza_ocena
{
        if (TerminalTest(stan)) then return Wynik(stan);
        if (Cutoff(stan, d)) then return Ocena(stan);
        v := + ∞ ;
        for (s ∈ Następne(stan)) do v := min(v, MaksOcena(s, d-1));
        return v;
}
```

9.5. Pytania

1. Wyjaśnić cel stosowania i zasady działania strategii „**przeszukiwanie MiniMax”**
2. **Omówić strategię „cięć α-β”.** Czy jest ona optymalna?
3. Wyjaśnić cel stosowania i zasady działania strategii **„obciętego Mini-max-u”.** Czy jest ona optymalna - jeśli tak, to w jakich warunkach a jeśli nie - to dlaczego?

9.6. Zadania

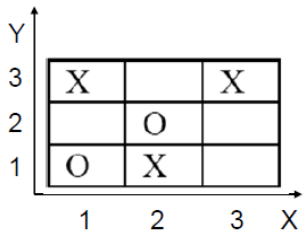
Zad. 9.1

Zilustrować zasady działania strategii:

A) „**przeszukiwanie Mini-Max”** i

B) **„cięcia α-β” ,**

na przykładzie „gry w kółko i krzyżyk”, w poniższej sytuacji:



Podać drzewa decyzyjne rozwijane w obu strategiach.

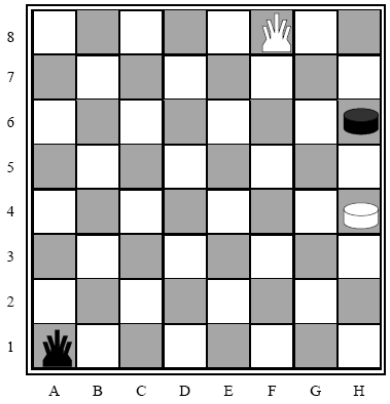
Zad. 9.2

Wyjaśnić cel stosowania i zasadę działania strategii **przeszukiwania „obciętego Mini-maksu”.**

Zilustrować odpowiedź na przykładzie „gry w warcaby”, w poniższej sytuacji:

- każdej ze stron została 1 damka i 1 pionek,
- białe podążają „w górę”,
- ruch należy do białych,
- **poziom obcinania** wynosi 2.

Zaproponować ocenę stanu gry.



Część III: Uczenie się poprzez indukcję

10. Uczenie się na podstawie obserwacji - indukcja

10.1. Uczenie się przez indukcję

10.1.1. Cel uczenia

Uczenie się przez indukcję polega na wnioskowaniu (nabywaniu wiedzy) na podstawie obserwacji (zakłada się w nim dostępność próbek uczących (inna nazwa: przykłady trenujące):

$T \Rightarrow h \wedge W$,

gdzie h – hipoteza wygenerowana w wyniku wnioskowania indukcyjnego, W - wiedza wrodzona, T – przykłady trenujące (próbki uczące), □ - logiczna implikacja (h, W, T są prawdziwe).

Indukcyjna hipoteza wraz z wiedzą wrodzoną (być może pustą) wyjaśnia możliwie dokładnie otrzymaną informację uczącą.

Celem indukcyjnego wnioskowania jest nie tylko odkrycie pewnych zależności w danych i wyjaśnieniu przykładów trenujących, ale przede wszystkim przewidywanie nowych faktów i obserwacji.

Założenie o indukcji: hipoteza wybrana na podstawie zbioru uczącego sprawdza się także dla przykładów spoza tego zbioru.

10.1.2. Pojęcia

Pojęcia są formą reprezentacji wiedzy. Pojęcie przypisuje obiektom etykiety ich kategorii.

Przez „pojęcie" rozumiemy podzbiór obiektów lub zdarzeń definiowanych w ramach pewnego większego zbioru.

Np. pojęcie „ptak" jest podzbiorem obiektów w zbiorze wszystkich rzeczy takich, które zaliczamy do kategorii „ptaków".

- Alternatywnie: „pojęcie" to funkcja charakterystyczna zbioru – przyjmuje wartość „True" dla przykładu należącego do kategorii i „False" w przeciwnym przypadku:

$$c(x) = \begin{cases} 1 & \text{gdy przykład x należy do kategorii (przykład pozytywny)} \\ 0 & \text{w przeciwnym przypadku (przykład negatywny)} \end{cases}$$

- Gdy kategorii jest więcej niż dwa mówimy wtedy o pojęciach wielokrotnych :

$$c(x) = c_i \text{ ,gdzie } c_i \in C$$

10.1.3. Zbiór treningowy (próbki uczące)

Oznaczmy: *T* - zbiór przykładów trenujących pochodzących z pewnej dziedziny *X*.

Wiedza zdobyta na ograniczonym zbiorze przykładów *T* ma być użyteczna do przewidywania właściwości przykładów *x* z całej dziedziny *X*.

Założymy, że dziedzina jest opisana za pomocą ustalonego zbioru atrybutów: $A = \{a_1, a_2, \ldots, a_n\}$, gdzie

$$a_1 : X \to A_1, \quad a_2 : X \to A_2, \quad \ldots, \quad a_n : X \to A_n.$$

Przykład (próbkę) utożsamimy z wektorem wartości atrybutów:

$$\{x \in X : x = [a_1(x) = v_1, a_2(x) = v_2, \ldots, a_n(x) = v_n,]\}$$

10.1.4. Sposoby uczenia

Wyróżnimy trzy podstawowe sposoby indukcyjnego uczenia:

1. **uczenie się pojęć** - uczenie się sposobu przydzielania obiektów do kategorii (klas) - uczenie się **klasyfikacji** - proces nadzorowanego uczenia;
2. **tworzenie pojęć** - uczenie się tworzenia klas (proces nienadzorowanego uczenia) i przydzielania do nich obiektów (czyli uczenie się pojęć),
3. **uczenie się aproksymacji funkcji** - uczenie się odwzorowywanie obiektów na liczby rzeczywiste (proces nadzorowanego uczenia).

10.1.5. Uczenie się pojęć

Uczenie się pojęć polega na budowie pewnej hipotezy h, która na podstawie wartości atrybutów przykładu potrafi estymować związane z nim pojęcie C :

$$h : X \mapsto C$$

Najczęściej hipoteza jedynie przybliża rzeczywiste kategorie w lepszym lub gorszym stopniu:

$$h(x) \approx c(x)$$

Uczenie polega na znalezieniu hipotezy, która możliwie najlepiej odzwierciedla pojęcie docelowe w zbiorze etykietowanych przykładów *T*. Wybór hipotezy ma minimalizować błąd rzeczywisty na całej dziedzinie, który estymuje się za pomocą błędu na zbiorze trenującym.

10.1.6. Tworzenie pojęć

Niech próbki (przykłady) uczące pozbawione są etykiety klasy, do której należą. W procesie tworzenia pojęć należy znaleźć zależności (podobieństwa cech) pomiędzy przykładami i pogrupować przykłady w klasy. Grupowanie (klasteryzacja, samoorganizacja) odbywa się na zasadzie minimalizowania różnic przykładów z jednej grupy i maksymalizowania różnić pomiędzy grupami.

W przypadku tworzenia pojęć hipoteza określa zarówno tworzone kategorie jak i przynależność przykładów do nich.

W tworzeniu pojęć zawiera się uczenie pojęć, gdyż po stworzeniu pojęć i klasyfikacji przykładów trenujących, należy jeszcze nauczyć się tych pojęć, by móc klasyfikować nowe przykłady.

10.1.7. Uczenie się aproksymacji

Uczenie się aproksymacji funkcji jest podobne do uczenia się pojęć z tą różnicą, że liczba kategorii jest zbiorem liczb rzeczywistych, a nie określonym z góry skończonym zbiorem jak w przypadku uczenia się pojęć.

- Próbka ucząca ma postać sekwencji argumentów funkcji oraz wartości funkcji dla tych argumentów, która odpowiada etykiecie kategorii.
- Celem uczenia jest aproksymacja postaci funkcji na podstawie przykładów na całą dziedzinę.

10.2. Tryb podawania próbek

Algorytmy indukcyjnego uczenia się można implementować jako strategie przeszukiwania przestrzeni hipotez:

1. poszukujemy dobrej hipotezy w pewnej przestrzeni możliwych pojęć czy funkcji, zdefiniowanej w języku wybranym dla tego zadania;
2. przestrzeń hipotez powinna zawierać wszystkie hipotezy jakie może skonstruować uczeń;
3. przynajmniej jedna z tych hipotez jest poprawna.

Definicja. Hipoteza jest **nadmiernie dopasowana** do zbioru uczącego, jeśli inna hipoteza o większym błędzie próbki na tym zbiorze ma mimo to mniejszy błąd rzeczywisty.

Aby ograniczyć ryzyko nadmiernego dopasowania należy preferować hipotezy proste o małym błędzie próbki na zbiorze uczącym.

Można ograniczyć liczbę pomyłek stosując odpowiedni *tryb podawania* próbek uczących.

10.2.1. Tryby podawania próbek uczących

Wyróżnimy następujące tryby podawania próbek:

1. wsadowy
2. inkrementacyjny
3. epokowy
4. korekcyjny

Tryb wsadowy

W trybie tym cały zbiór trenujący jest od razu dostępny, a po jego przetworzeniu podawana jest hipoteza. Powiększenie zbioru trenującego (pojawienie się nowych przykładów) powoduje konieczność rozpoczęcia uczenia się od nowa.

Tryb inkrementacyjny

Uczeń otrzymuje na raz tylko jeden przykład i zgodnie z nim zmienia swoją hipotezę. W każdej chwili uczenia dostępna jest hipoteza uznawana za ostateczną, gdy wyczerpią się próbki.

Tryb epokowy

Proces uczenia zorganizowany jest w cykle zwane epokami, w każdej z których jest przetwarzany pewien zbiór przykładów trenujących. Gdy epoka składa się jedynie z jednej próbki to otrzymujemy tryb inkrementacyjny.

Tryb korekcyjny

Tryb korekcyjny ma zastosowanie w przypadku uczenia się z nadzorem (w zadaniach uczenia pojęć lub aproksymacji). Na każdym etapie uczenia istnieje aktualna hipoteza.

Kroki

1. Uczeń otrzymuje przykład nieetykietowany, wyznacza dla niego kategorię (lub wartość funkcji docelowej) stosując swoją aktualną hipotezę.
2. Odpowiedź ucznia jest porównywana z etykietą przykładu, po czym przekazuje się mu informację korekcyjną (poprawną kategorię przykładu lub poprawną wartość aproksymowanej funkcji).
3. Hipoteza jest modyfikowana odpowiednio do wartości korekcji.

10.2.2. Model ograniczania pomyłek

Model ograniczania liczby pomyłek zakłada przyjęcie korekcyjnego trybu uczenia się. Uczeń przewiduje kategorię $h(\mathbf{x})$ dla każdego przykładu trenującego $\mathbf{x}$ stosując wybraną przez siebie hipotezę h, a dopiero później poznaje poprawną kategorię $c(\mathbf{x})$.

Definicja. Dla klasy pojęć C i algorytmu uczenia się L ograniczenie liczby pomyłek $M_L(C)$ to maksymalna liczba pomyłek, jakie uczeń używający algorytmu L i przestrzeni hipotez, $H = C$, popełni w najgorszym przypadku ucząc się dowolnego pojęcia z C.

Definicja. Dla dowolnej klasy pojęć C optymalne ograniczenie liczby pomyłek to minimalna wartość ograniczenia liczby pomyłek dla wszystkich możliwych algorytmów uczenia się:

$$Opt(C) = \min_L M_L(C)$$

10.3. Uczenie się pojęć

Uczenie się pojęcia: automatyczne wnioskowanie definicji „pojęcia” na podstawie zbioru próbek, etykietowanych jako przynależnych lub nie przynależnych do „pojęcia”.

Innymi słowy: uczenie się pojęcia polega na wyznaczeniu boolowskiej funkcji (klasyfikatora) zdefiniowanej na przestrzeni atrybutów obiektu, na podstawie zadanych próbek uczących podających wejście i wyjście tej funkcji.

Pojęcie - podzbiór obiektów X: $c: X \rightarrow \{0, 1\}$

Funkcja docelowa. Np. $H: \text{Niebo} \times \text{Temperatura} \times \text{Wilgotność} \times \text{Wiatr} \times \text{Woda} \times \text{Prognoza} \rightarrow \{\text{Tak, Nie}\}$

10.3.1. Własności hipotezy

Hipoteza - charakterystyka obiektów przynależnych do pojęcia w postaci ograniczeń nakładanych na atrybuty obiektu:

$h: attr(X) \rightarrow \{0, 1\}$

Spełnialność

h(x) = 1, wtw. atrybuty x spełniają wszystkie ograniczenia zadane jako h;

h(x) = 0, w przeciwnym razie.

Zgodność

h(x) = c(x), dla każdego obiektu x ze zbioru treningowego.

Poprawność:

h(x) = c(x), dla każdego obiektu x z X .

Celem uczenia jest znalezienie zgodnej i poprawnej hipotezy, czyli takiego h, że: h(x)=c(x), dla wszystkich x w X.

Przykład 1: pogoda sprzyjająca uprawianiu sportów wodnych.

- **Pojęcie:** właściwe dni dla uprawiania sportów wodnych (wartości: Tak, Nie)
- **Atrybuty / cechy:**
  - Niebo (wartości: jasne, zachmurzone, deszczowe)
  - Temperatura powietrza (wartości: ciepło, chłodno)
  - Wilgotność (wartości: normalna, wysoka)
  - Wiatr (wartości: silny, słaby)
  - Woda (wartości: ciepła, zimna)

- Prognoza (wartości: stała, zmiana)

- **Przykład próbki:**

<słoneczne, ciepło, wysoka, silny, ciepła, stała, Tak>

Zbiór próbek:

Dzień	Niebo	Tempe-ratura	Wilgotność	Wiatr	Woda	Prognoza	Sporty wodne
1	Jasne	Ciepło	Normalna	Silny	Ciepła	Stała	Tak
2	Jasne	Ciepło	Wysoka	Silny	Ciepła	Stała	Tak
3	Deszcz	Chłodno	Wysoka	Silny	Ciepła	Zmiana	Nie
4	Jasne	Ciepło	Wysoka	Silny	zimna	Zmiana	Tak

Reprezentacja hipotezy - **koniunkcja ograniczeń dla każdego atrybutu**:

- "?" : "możliwa jest dowolna wartość"
- "Ø" : "żadna wartość nie jest możliwa"
- konkretna wartość.

Przykład hipotezy: h = <?, zimno, wysoka, ?, ?, ?> ,

czyli chłodny dzień i wysoka wilgotność są dobrą pogodą dla sportów wodnych.

10.3.2. Przestrzeń wersji

Algorytm uczenia może mieć postać **przeszukiwania *przestrzeni wersji***.

Przestrzeń wersji jest podzbiorem hipotez zgodnych z przykładami uczącymi (jednym lub więcej), uporządkowanym relacją generalizacji hipotezy.

Strategia przeszukiwania może korzystać z uporządkowania hipotez w przestrzeni wersji (za pomocą **relacji generalizacji**) i realizować ukierunkowane przeszukiwanie przestrzeni. Generalizacja następuje dzięki operacjom:

1. Zastąpienie stałych zmiennymi,
2. Wyeliminowanie warunku z koniunkcji,
3. Dodanie warunku do wyrażenia,
4. Zastąpienie cechy cechą nadrzędną w hierarchii klas.

Cel przeszukiwania: znaleźć „najlepszy" opis pojęcia ze zbioru wszystkich możliwych wersji (hipotez); "najlepszy" oznacza taki, który najlepiej uogólnia wszystkie (znane lub nieznane) elementy przestrzeni atrybutów obiektów.

Najbardziej ogólna hipoteza – wszystkie wartości są dozwolone. Np. <?, ?, ?, ?, ?, ?>

Najbardziej specyficzna hipoteza – żadne wartości nie są dozwolone. Np. < Ø, Ø, Ø, Ø, Ø, Ø >

10.3.3. Algorytm „Find S”

Algorytm „Find S” poszukuje najbardziej specyficznej hipotezy zgodnej z przykładami trenującymi (tab. 10-1)

Tab. 10-1: Algorytm „Find S”.

1. Inicjalizacja: h – najbardziej specyficzna hipoteza w H
2. FOR każdy „pozytywny” przykład trenujący x:
3. FOR każde ograniczenie atrybutu a_i w h:
4. IF ograniczenie nie jest spełnione przez x THEN zamień a_i przez następne bardziej ogólne ograniczenie spełnione przez x
5. RETURN: hipoteza h

Przykład 2.

Dla poniższego zestawu przykładów algorytm „Find S” poszukuje najbardziej specyficznej hipotezy zgodnej z pozytywnymi przykładami :

Dzień	Niebo	Tempe-ratura	Wilgotność	Wiatr	Woda	Prognoza	Sporty wodne?
1	Jasne	Ciepło	Normalna	Silny	Ciepła	Stała	Tak
2	Jasne	Ciepło	Wysoka	Silny	Ciepła	Stała	Tak
3	Deszcz	Chłodno	Wysoka	Silny	Ciepła	Zmiana	Nie
4	Jasne	Ciepło	Wysoka	Silny	Zimna	Zmiana	Tak

Kolejno pobierane są pozytywne próbki uczące i modyfikowana jest hipoteza:

1. Init → $h = < \varnothing , \varnothing , \varnothing , \varnothing , \varnothing , \varnothing >$
2. Przykład 1 → $h =$ <jasne, ciepło, normalna, silny, ciepła, stała>
3. Przykład 2 → $h =$ <jasne, ciepło, ? , silny, ciepła, stała>
4. Przykład 4 → $h =$ <jasne, ciepło, ? , silny, ? , ? >

Wynik jest zgodny z pozytywnymi przykładami, a jeśli przykłady są poprawne to jest także zgodny z negatywnymi przykładami.

Algorytm „Find S” jest łatwy do implementacji, ale posiada istotne wady:

- Nie wiadomo czy znaleziona hipoteza jest jedyną możliwą.
- Czy preferowanie najbardziej specyficznej hipotezy jest właściwym rozwiązaniem? Przecież zbiór uczący może być niezgodny (zaszumiony).
- Jeśli jest więcej równie specyficznych hipotez zgodnych ze zbiorem uczącym, algorytm ich wszystkich nie znajdzie, gdyż nie przewiduje „nawrotu”.

10.3.4. Algorytm Eliminacji Kandydatów

Algorytm Eliminacji Kandydatów (CEA: *Candidate Elimination Algorithm*) realizuje przeszukiwanie *przestrzeni wersji* jednocześnie w dwóch kierunkach wyznaczonych przez relację generalizacji hipotez: przechodząc od szczegółu do ogółu i na odwrót.

Algorytm **CEA** redukuje aktualny rozmiar przestrzeni wersji wraz z dodaniem kolejnych przykładów.

Algorytm CEA stosuje zwartą reprezentację przestrzeni wersji: zamiast przeliczać wszystkie hipotezy zgodne z przykładami trenującymi, można reprezentować ich **najbardziej specyficzne i najbardziej ogólne ograniczenie** (kontur), a hipotezy zawarte pomiędzy tymi ograniczeniami generować tylko w miarę potrzeby.

Ogólnym ograniczeniem *G*, ze względu na przestrzeń hipotez *H* i dane trenujące *D*, jest zbiór maksymalnie ogólnych hipotez w *H* zgodnych z *D*.

Specyficznym ograniczeniem *S*, ze względu na przestrzeń hipotez *H* i dane trenujące *D*, jest zbiór minimalnie ogólnych (tzn. maksymalnie specyficznych) hipotez w *H* zgodnych z *D*.

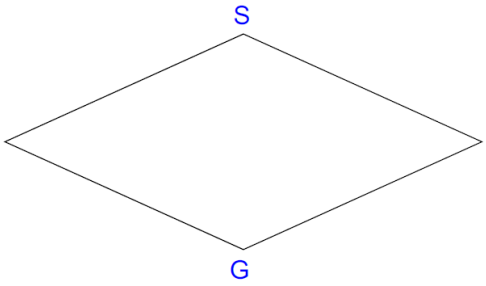
Niech G będzie zbiorem hipotez maksymalnie ogólnych, zaś S zbiorem hipotez maksymalnie szczegółowych. Algorytm CEA na przemian uszczegółowia G i uogólnia S, aż zbiegną się one do jednego pojęcia docelowego (tab. 10-2).

Tab. 10-2: Implementacja algorytmu CEA

```
funkcja CEA(przykłady) zwraca pojedyncze pojęcie
{
  Przypisz G najogólniejsze pojęcie (hipotezę) z przestrzeni;
  Przypisz S pierwszy pozytywny przykład uczący;
  for każdy nowy pozytywny przykład p {
     Usuń z G hipotezy niezgodne z p ;
     for każde s ∈ S
       if (s zgodne z p) then pozostaw;
       else zastąp s jej wszystkimi najbliższymi uogólnieniami
                h zgodnymi z p i takimi, że dla każdego nowego h
                istnieje hipoteza bardziej ogólna w G ;
     Usuń z S takie hipotezy, które są ogólniejsze od innej
       hipotezy w S ;
  }
  for każdy nowy negatywny przykład n {
    Usuń z S pojęcia niezgodne z n ;
    for (dla każdego g ∈ G)
      if g zgodne z n then zostaw;
      else zastąp g najbliższymi bardziej specyficznymi
        hipotezami h zgodnymi z n i dla których istnieją w S
        hipotezy bardziej specyficzne;
     Usuń z G takie hipotezy, które są bardziej specyficzne niż
       inna hipoteza w G;
  }
  // Jeśli ((G = = S) ∧ (G i S zawierają jedno pojęcie))
  // to G = S jest unikalnie zadanym znalezionym pojęciem.
  return G;
}
```

Reprezentując jedynie ograniczenia hipotez uzyskujemy zwartą reprezentację przestrzeni wersji:

$$<G, S> = \{h \in H \mid \exists g \in G\ \exists s \in S:\ g \geq_g h \geq_g s\}$$



Przykład 3.

Zastosujemy algorytm eliminacji kandydatów do nauczenia się pojęć w naszym przykładzie pogody nad morzem sprzyjającej wodnym sportom (próbki uczące jak w przykładzie 2).

Inicjalizujemy dwa zbiory:

- G_0 - zbiór maksymalnie ogólnych hipotez w przestrzeni hipotez tego problemu H,
- S_0 – zbiór najbardziej specyficznych hipotez w przestrzeni H (efektywniejsze rozwiązanie – pierwsza pozytywna próbka).

W tym przypadku:

- $S_0 = \{<\varnothing, \varnothing, \varnothing, \varnothing, \varnothing, \varnothing>\}$ lub (lepiej)

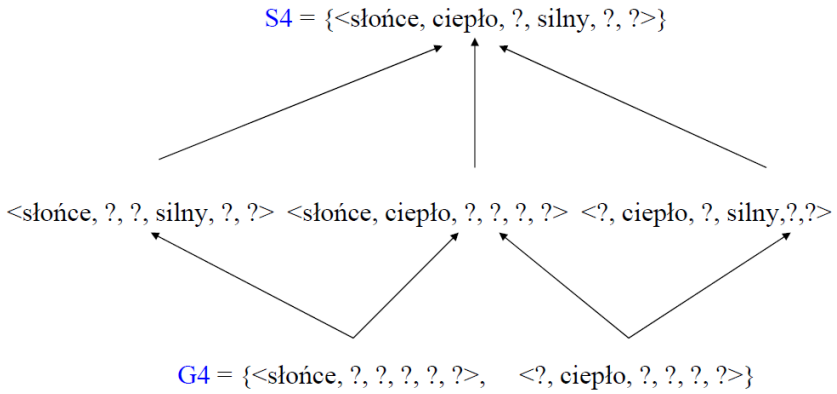
$S_0 = \{<$słońce, ciepło, normalna, silny, ciepła, stała$>$

- $G_0 = \{<?, ?, ?, ?, ?, ?>\}$

Kolejne iteracje:

- S1 = {<słońce, ciepło, normalna, silny, ciepła, stała>}
- G1 = {<?, ?, ?, ?, ?, ?>}
- S2 = {<słońce, ciepło, ?, silny, ciepła, stała>}
- G2 = {<?, ?, ?, ?, ?, ?>}
- S3 = {<słońce, ciepło, ?, silny, ciepła, stała>}
- G3 = {<słońce, ?, ?, ?, ?, ?>,
  <?, ciepło, ?, ?, ?, ?>,
  <?, ?, ?, ?, ?, stała> }

W kolejnej iteracji G4 zdąża do S4 i proces kończy się w sytuacji, gdy (G = S = S4).



Przestrzeń wersji w algorytmie CEA zbiega do poprawnego pojęcia jeśli:

- Przestrzeń hipotez H zawiera szukane pojęcie;
- Próbki uczące nie zawierają błędów.

Efektywna strategia pobierania próbek: kolejna wymagana próbka ucząca powinna rozdzielać alternatywne hipotezy zawarte w aktualnej przestrzeni wersji.

Jeśli pojęcie jest nauczone jedynie częściowo (istnieją alternatywne hipotezy) to można zastosować wielokrotną klasyfikację w oparciu o każda hipotezę a następnie ustalić klasę poprzez **głosowanie większościowe**.

10.4. Pytania

1. Wymienić i omówić istotę trzech podstawowych sposobów indukcyjnego uczenia.
2. Omówić tryby podawania próbek uczących.
3. Na czym polega przestrzeń wersji?
4. Omówić algorytm „Find-S”.
5. Przedstawić algorytm eliminacji kandydatów.

10.5. Zadania

Zad. 10.1

Zastosować algorytm eliminacji kandydatów do **nauczenia się pojęć** umożliwiających binarną klasyfikację pogody nad morzem na sprzyjającą wodnym sportom lub nie. Przyjąć, że próbka ucząca jest agregacją następujących obserwacji:

<niebo, temperatura, wilgotność, wiar, woda, prognoza>.

Zadane są następujące próbki uczące (3 pozytywne i 1 negatywna):

Próbka	Niebo	Temp.	Wilgotność	Wiatr	Woda	Prognoza	Sprzyja?
1	Słońce	Ciepło	Normalna	Silny	Ciepła	Bez zmian	Tak
2	Słońce	Ciepło	Wysoka	Silny	Ciepła	Bez zmian	Tak
3	Deszcz	Zimno	Wysoka	Silny	Ciepła	Zmiana	Tak
4	Słońce	Ciepło	Wysoka	Silny	Chłodna	Zmiana	Nie

11. Uczenie się klasyfikacji

11.1. Zadanie klasyfikacji

Zadanie klasyfikacji polega na znalezieniu kategorii (klasy) dla zadanego przykładu (próbki uczącej, obserwacji). Przykłady trenujące (próbki uczące) mają postać par:

- opis obiektu i
- etykieta jego kategorii (klasy).

Taką parę nazywamy przykładem etykietowanym.

Przykładem nieetykietowanym jest taki, który obejmuje jedynie opis obiektu.

Jako przykład zadania klasyfikacji omówimy tu uczenie drzew decyzyjnych. Tu jedynie zastanowimy się, skąd wiemy czy dana klasa pojęć jest **nauczalna**.

Klasę pojęć nazywamy **nauczalną**, jeżeli istnieje efektywny algorytm znajdujący z dużym prawdopodobieństwem pojęcie „w przybliżeniu poprawne" – co oznacza, że pojęcie to poprawnie klasyfikuje duży procent możliwych przykładów.

Skąd mamy wiedzieć czy dany algorytm uczący wygenerował hipotezę która poprawnie sklasyfikuje przyszłe dane? Odpowiedź na to pytanie daje nam **model PAC** (*probably approximately correct*), wykorzystujący metody wnioskowania statystycznego. Zasada na której opiera się całe rozumowanie to: hipoteza zgodna z wystarczająco dużym zbiorem trenującym jest prawdopodobnie w przybliżeniu poprawna.

11.2. Drzewa decyzyjne

W drzewie decyzyjnym występują:

- **węzły** niekońcowe – odpowiadają one testom przeprowadzanym na atrybutach przykładów,
- **gałęzie** (krawędzie) – odpowiadają one możliwym wynikom tych testów,
- **liście** (węzły końcowe) – odpowiadają one etykietom kategorii (klas).

Klasyfikacja przykładu za pomocą drzewa decyzyjnego polega na przejściu ścieżki od korzenia do liścia drzewa wzdłuż gałęzi wyznaczanych przez wyniki testów związanych z odwiedzanymi kolejno węzłami. Osiągnięcie liścia wyznacza kategorię (klasę) dla tego przykładu – drzewo decyzyjne reprezentuje **hipotezę**.

Przykład 1. Problem decyzyjny: czy czekać na wolny stolik w restauracji? Załóżmy, że dla podjęcia decyzji korzystamy z następujących **atrybutów**:

- **Alternatywa**: czy jest inna restauracja w okolicy?
- **Bar**: czy można oczekiwać w barze?
- **Piątek**: czy dziś jest piątek czy już sobota?
- **Głód**: czy jesteśmy głodni?
- **Osoby**: liczba osób w restauracji (nikogo, trochę, pełno)
- **Cena**: zakres cenowy (+, ++, +++)

- **Deszcz**: czy pada deszcz?
- **Rezerwacja**: czy mamy rezerwację?
- **Typ**: rodzaj restauracji (francuska, włoska, tajlandzka, *burger*)

Czas: szacowany czas czekania (0-10 min.,10-30, 30-60, >60).

Próbki uczące (przykłady trenujące) są opisane poprzez wartości atrybutów, które w naszym przykładzie 1 mogą być typu logicznego, dyskretne (liczby całkowite) lub ciągłe. Na potrzeby drzewa decyzyjnego ciągła dziedzina zostaje podzielona na przedziały.

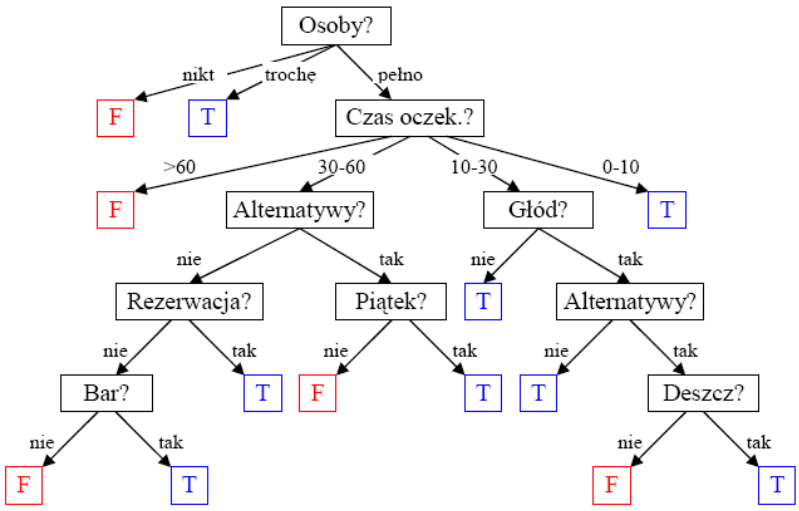
Próbki uczące w przykładzie 1 opisują sytuacje, w których będziemy czekali lub nie czekali na stolik. Klasyfikujemy każdy przykład (decyzja) na jedną z dwóch klas (tab.11-1):

- pozytywną – decyzja „*czekać*" (T) lub
- negatywną – decyzja „*nie czekać*" (F).

Tab. 11-1. Próbki uczące w przykładzie 1

Próbka	Atrybuty										CEL
	Alt	Bar	Piąt	Głód	Osoby	Cena	Deszcz	Rez	Typ	Czas	Czekaj
X1	Tak	Nie	Nie	Tak	Trochę	+++	Nie	Tak	franc	0-10	True
X2	Tak	Nie	Nie	Tak	Pełno	+	Nie	Nie	tajski	30-60	False
X3	Nie	Tak	Nie	Nie	Trochę	+	Nie	Nie	burger	0-10	True
X4	Tak	Nie	Tak	Tak	Pełno	+++	Nie	Nie	tajski	10-30	True
X5	Tak	Nie	Tak	Nie	Pełno	+++	Nie	Tak	franc	>60	False
X6	Nie	Tak	Nie	Tak	Trochę	++	Tak	Tak	włoski	0-10	True
X7	Nie	Tak	Nie	Nie	Nikt	+	Tak	Nie	burger	0-10	False
X8	Nie	Nie	Nie	Tak	Trochę	++	Tak	Tak	tajski	0-10	True
X9	Nie	Tak	Tak	Nie	Pełno	+	Tak	Nie	burger	>60	False
X10	Tak	Tak	Tak	Tak	Pełno	+++	Nie	Tak	włoski	10-30	False
X11	Nie	Nie	Nie	Nie	Nikt	+	Nie	Nie	tajski	0-10	False
X12	Tak	Tak	Tak	Tak	Pełno	+	Nie	Nie	burger	30-60	True

Przykład drzewa decyzyjnego dla tego problemu pokazano na rys. 11.1



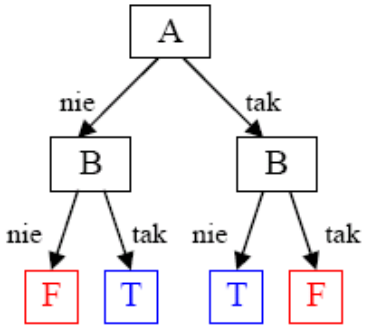
Rys. 11.1 Przykład drzewa decyzyjnego dla problemu „czekanie w restauracji".

11.2.1. Siła wyrazu drzew decyzyjnych

Drzewa decyzyjne mogą wyrazić każdą **funkcję logiczną lub dyskretną** zależną od zbioru atrybutów wejściowych. Np. dla funkcji logicznych (o wartościach typu Boolean) pojedynczy wiersz w tabeli prawdy odpowiada pojedynczej ścieżce w drzewie od korzenia do liścia (rys. 11.2).

A	B	A ⊗ B
F	F	F
F	T	T
T	F	T
T	T	F

⊗ oznacza funkcję „XOR"



Rys.11.2 Funkcja logiczna (lewa strona) i jej reprezentacja w postaci drzewa decyzyjnego (prawa strona).

Jaka jest liczba możliwych drzew decyzyjnych dla *n* atrybutów typu logicznego (Boolean)? Każda funkcja logiczna o *n* binarnych atrybutach odpowiada tablicy prawdy o 2^n wierszach. Liczba różnych tablic prawdy o takim rozmiarze wynosi 2^{2n}. Np. dla n=6 istnieje 18,446,744,073,709,551,616 drzew. Tak duża przestrzeń problemu uczenia wymaga stosowania nietrywialnych algorytmów uczenia drzew decyzyjnych.

11.2.2. Uczenie się drzewa decyzyjnego

Problem uczenia się drzewa decyzyjnego

1. Dla deterministycznej funkcji zmiennych *X* zawsze można utworzyć drzewo decyzyjne w pełni zgodne z próbkami uczącymi takie, że każdej próbce odpowiadać będzie pojedyncza ścieżka od korzenia do liścia.
2. Jednak drzewo z pkt. 1 zwykle nie będzie dobrze uogólniać (przewidywać) nieznanych próbek. Dlatego należy znaleźć bardziej zwarte drzewo decyzyjne.

Idea zstępującej metody uczenia drzewa decyzyjnego

1. Cel – szukamy zwartego (niedużego) drzewa zgodnego z próbkami uczącymi.
2. Postępowanie – rekursywnie wybieramy "najważniejszy" atrybut do roli korzenia poddrzewa.

Algorytm DTL uczenia się drzewa decyzyjnego, zbudowany w oparciu o powyższą ideę, przedstawia tab. 11-2.

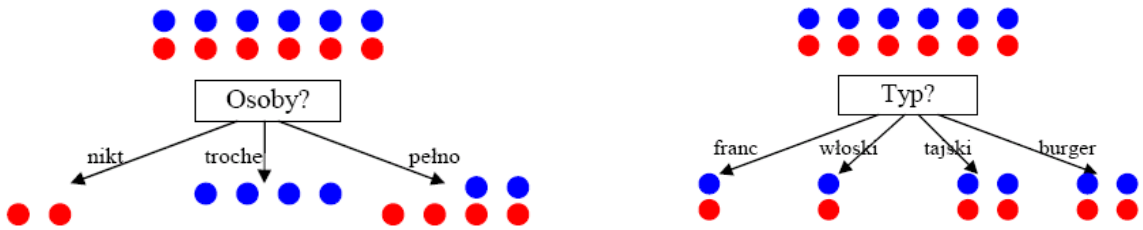
Tab. 11-1 Implementacja strategii uczenia się drzewa decyzyjnego

```
function DTL(próbki, atrybuty, domyślna) returns drzewo decyzyjne
{ if (próbki == ∅) then return domyślna;
    else if (wszystkie próbki należą do jednej klasy)
        then return KLASA(próbki);
        else if (atrybuty == ∅)
            then return KLASAWIĘKSZOŚCIOWA(próbki);
            else
            {   best ← WYBIERZATRYBUT(atrybuty, próbki);
                drzewo ← nowe drzewo o korzeniu testującym atrybut best ;
                m ← KLASAWIĘKSZOŚCIOWA(próbki);
                for each (wartość v_i ∈ best) do
                {   próbki_i ← {elementy w próbki o best = v_i};
                    poddrzewo ← DTL(próbki_i, atrybuty - best, m);
                    dodaj gałąź do drzewa o etykiecie v_i i poddrzewie poddrzewo;
                }
            }
            return drzewo;
}
```

Realizacja podfunkcji WYBIERZATRYBUT w algorytmie DTL.

Wybór następuje według zasady: “dobry” atrybut to taki, który dzieli próbki na podzbiory, które są (w idealnym przypadku) “wszystkie pozytywne” lub “wszystkie negatywne”.

Przykład 2. Dla próbek z przykładu 1 testując atrybut *Osoby* (rys. 11.3 - lewa strona) zbliżamy się do rozdzielenia przypadków pozytywnych od negatywnych. Testowanie atrybutu *Typ* (rys. 11.3 - prawa strona) zasadniczo niczego w tym zakresie nie wnosi.



Rys.11.3. Alternatywne drzewa tworzone przy testowaniu dwóch różnych atrybutów

11.2.3. Kryteria wyboru testów

Formalnie biorąc implementacja podfunkcji **WYBIERZATRYBUT** oparta może być na teorii informacji.

Oczekiwana wartość informacji (**entropia**) dla zmiennej losowej X o n wartościach wynosi:

$$H(P_X(v_1), \ldots, P_X(v_n)) = - \Sigma_{i=1}^{n} [P_X(v_i) \log_2 P_X(v_i)]. \qquad (11\text{-}1)$$

Im bardziej prawdopodobna realizacja zmiennej losowej tym mniej informacji ona zawiera. Z kolei im większa niepewność zdarzenia tym więcej zawiera ono informacji. Dla zmiennej losowej o dwóch równie prawdopodobnych możliwych wartościach ([0.5, 0.5]) entropia wynosi 1 [bit].

Możemy oceniać “jakość” danego atrybutu na podstawie ilości informacji, jaka pozostanie, tzn. będzie jeszcze zawarta w zbiorze próbek po jego testowaniu zadanym atrybutem.

Niech zbiór próbek **dwóch klas** zawiera p próbek dla klasy „pozytywnej” i n dla „negatywnej”. Wtedy zawartość informacyjna tego zbioru próbek („przed testowaniem”) wynosi:

$$\mathrm{H}(\frac{p}{p+n},\frac{n}{p+n}) = -\frac{p}{p+n}\log_2\frac{p}{p+n} - \frac{n}{p+n}\log_2\frac{n}{p+n} \tag{11-2}$$

Wybrany atrybut A dzieli zbiór próbek E na podzbiory, $E_1,\ldots,E_v$, odpowiednio do przyjmowanych przez nie wartości dla A, gdy A ma v różnych wartości. Oczekujemy, że każdy z podzbiorów będzie zawierał już mniej informacji po testowaniu ze względu na atrybut A niż przedtem.

W każdym podzbiorze będzie p_i próbek o klasie pozytywnej i n_i próbek o klasie negatywnej. Oczekiwana wartość pozostałej entropii to ważona suma entropii dla każdego podzbioru:

$$\mathrm{H}_{pozostało}(A) = \sum_{i=1}^{v}\frac{p_i+n_i}{p+n}\cdot\mathrm{H}(\frac{p_i}{p_i+n_i},\frac{n_i}{p_i+n_i}) \tag{11-3}$$

Tym samym zysk informacji (ZI) odpowiadający redukcji entropii po wykonaniu testu dla atrybutu wynosi:

$$\mathrm{ZI}(A) = \mathrm{H}(\frac{p}{p+n},\frac{n}{p+n}) - \mathrm{H}_{pozostało}(A) \tag{11-4}$$

W podfunkcji **WYBIERZATRYBUT** stosujemy więc zasadę wyboru atrybutu o **największym zysku informacji**:

$$\arg\max_A \mathrm{ZI}(A) \tag{11-5}$$

Przykład 3.

Dla zbioru próbek z przykładu 2 zachodzi: $p = n = 6;\ \mathrm{H}([6/12, 6/12]) = 1$ bit.

Określimy zysk informacji dla atrybutów *Osoby* i *Typ* testowanych na podanym zbiorze:

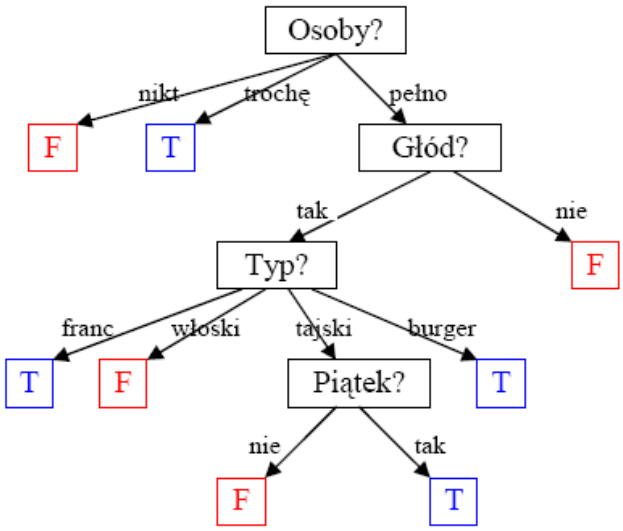
$$\mathrm{ZI}(Osoby) = 1 - [\frac{2}{12}\mathrm{H}([0,1]) + \frac{4}{12}\mathrm{H}([1,0]) + \frac{6}{12}\mathrm{H}([\frac{2}{6},\frac{4}{6}])] = 0.5409\,[\text{bit/znak}]$$

$$\mathrm{ZI}(Typ) = 1 - [\frac{2}{12}\mathrm{H}([\frac{1}{2},\frac{1}{2}]) + \frac{2}{12}\mathrm{H}([\frac{1}{2},\frac{1}{2}]) + \frac{4}{12}\mathrm{H}([\frac{2}{4},\frac{2}{4}]) + \frac{4}{12}\mathrm{H}([\frac{2}{4},\frac{2}{4}])] = 0$$

Można sprawdzić, że atrybut *Osoby* zapewnia najwyższy zysk informacji ZI spośród atrybutów i zostanie on wybrany przez algorytm DTL do testowania próbek w korzeniu drzewa.

Przykład 4.

Na rys. 11.5 pokazano drzewo decyzyjne dla problemu z przykładu 1, powstałe w wyniku uczenia przez indukcję algorytmem DTL, przy kryterium maksymalizacji zysku entropii, dla zbioru 12 podanych próbek.



Rys. 11.5. Drzewo decyzyjne dla problemu z przykładu 1

11.3. Klasyfikator numeryczny

11.3.1. Definicja problemu

Klasyfikacja wektora cech (wartości zmiennych obserwowanych) polega na podjęciu decyzji zgodnie z **funkcją decyzyjną** dla tego klasyfikatora i wiedzy nabytej uprzednio na podstawie zbioru uczącego (w procesie **uczenia**). Klasyfikator **numeryczny** polega na zastosowaniu funkcji decyzyjnej, która przypisuje wektor cech (obserwacji) $\mathbf{c} \in \Re^d$ do dyskretnej klasy (modelu) Ω_κ:

$$\zeta(\mathbf{c}) = \begin{cases} \Re^d \rightarrow \{1,2,\cdots,K\} \\ \mathbf{c} \rightarrow \kappa \end{cases} \tag{11.6}$$

Ocena jakości i porównywanie różnych klasyfikatorów może uwzględniać szereg aspektów: wartość prawdopodobieństwa błędnej klasyfikacji, względnie koszt ryzyka, złożoność obliczeniowa funkcji decyzyjnej, zdolność do uogólnienia dla nieznanych próbek, zdolność do uczenia funkcji decyzyjnej, itd. Bardzo trudnym teoretycznym problemem jest określenie funkcji decyzyjnej, która optymalizuje wszystkie możliwe kryteria jakości klasyfikatora.

Sensowne jest przyjęcie podstawowych założenia przy konstrukcji klasyfikatora:

1. jeśli 2 wektory cech należą do tej samej klasy to występuje pomiędzy nimi mała odległość,
2. jeśli wektory cech należą do różnych klas to pomiędzy nimi jest duża odległość.

Miarami odległości mogą być: funkcje gęstości prawdopodobieństwa, odległość *Euklidesa*, metryka modułowa, odległość *Mahalanobisa*, itd.

Optymalny klasyfikator

Klasyfikator stochastyczny tworzony jest według następującej zasady: zgromadź założenia (wiedzę) o analizowanej dziedzinie; wyznacz ryzyko błędnej klasyfikacji (lub decyzji), wyznacz funkcję decyzyjną, która minimalizuje podane ryzyko.

Każda decyzja generuje pewne koszty (ryzyko). Po wielu decyzjach możemy określić średni koszt lub ryzyko. Obliczenie średnich wartości ryzyka wymaga istnienia pełnej informacji statystycznej (w postaci zadanego rozkładu zmiennej losowej) lub uprzedniej obserwacji dającej reprezentatywny zbiór próbek.

Założenia o analizowanej dziedzinie

Statystyczne własności klas reprezentowane są przez rodzinę n-wymiarowych rozkładów prawdopodobieństwa cech pod warunkiem klas: $p(\mathrm{c} \mid \Omega_k)$, k=1,...,K. Rozkład ten może mieć zadaną postać (parametryczny rozkład) lub nie (nieparametryczny). W procesie *uczenia* szacowane są parametry rozkładów względnie aproksymowane są całościowe rozkłady prawdopodobieństw w przestrzeni cech. Znany jest też rozkład prawdopodobieństwa klas ($p(\Omega_k)$, k=1,...,K).

Obliczenie ryzyka V

Oznaczamy koszty $\{r_{jk} = r\ (j \mid k)\ (j, k= 1,2, \ldots ;K)\}$ błędnej klasyfikacji, w wyniku której obiekt klasy o indeksie k jest błędnie klasyfikowany jako należący do klasy o indeksie j. Zakładamy, że koszty te są określane przez eksperta w zależności od zastosowania. Należy też oszacować prawdopodobieństwa błędnych decyzji, $p(j \mid k)$, j,k=1,...,K.

Teraz można już obliczyć sumaryczne ryzyko według wzoru:

$$\mathrm{V} = \sum_k \sum_j p_k\ p(j \mid k)\ r_{\mathrm{jk}}\ . \tag{11.7}$$

Reguła decyzyjna

Celem reguły decyzyjnej jest minimalizacja kosztu ryzyka V. Dla wyznaczenia ogólnej reguły należy przyjąć pewne założenia odnośnie funkcji kosztu.

Np. niech funkcja kosztu ma binarny charakter:

$$r_{kk} = 0, \qquad r_{ik} = 1, \text{ dla } \ i \neq k, \text{ i } \ i,k = 1, \ldots , K.$$

Taka postać funkcji kosztu wyznacza główną rolę w minimalizowaniu kosztu ryzyka V funkcji prawdopodobieństwa błędnej klasyfikacji. Dualnie, zamiast minimalizować prawdopodobieństwo błędnej klasyfikacji, możemy zdefiniować regułę decyzyjną, która maksymalizuje prawdopodobieństwo poprawnej klasyfikacji dla każdej obserwacji. W naturalny sposób wyraża to prawdopodobieństwo warunkowe a posteriori $p(\Omega_i \mid \mathrm{c})$. Czyli reguła decyzyjna optymalnego klasyfikatora stochastycznego obserwacji c przyporządkowuje klasę Ω_i o największej wartości

$$p(\Omega_i \mid \mathrm{c}) = \mathrm{p}(\ \mathrm{c}\mid \Omega_i)\ \mathrm{p}(\Omega_i),\ ,\ i = 1; \ldots , K.$$

Taką regułę decyzyjną stosuje *klasyfikator Bayesa*. Dla binarnej funkcji kosztu jest to optymalny klasyfikator w sensie minimalizacji *V*, a każdy dobrze określony klasyfikator numeryczny aproksymuje z góry błąd klasyfikatora Bayesa.

Jednak zastosowanie klasyfikatora Bayesa nie zawsze jest możliwe, gdyż wymaga on pełnej informacji statystycznej o analizowanej dziedzinie.

11.3.2. Podstawowe rodzaje klasyfikatorów numerycznych

Klasyfikator według funkcji potencjału („maszyna liniowa")

Należy określić postać parametrycznej funkcji (np. liniowa funkcja, wielomian 2-go stopnia) właściwej dla charakteru zbioru próbek (np. liniowo separowalne reprezentacje – cechy - wzorców różnych klas). Następnie w wyniku procesu uczenia klasyfikatora należy związać z każdą klasą jej konkretną funkcję wyrażającą tzw. potencjał w przestrzeni cech (uzyskujemy właściwe wartości parametrów funkcji potencjału dla każdej klasy). Wreszcie podczas aktywnej pracy klasyfikatora należy klasyfikować kolejne wektory cech zgodnie z kryterium maksymalizacji wartości funkcji potencjału dla danego wektora cech. Klasyfikator o liniowych funkcjach potencjału zwykle optymalizuje kryterium złożoności obliczeniowej dla klasyfikatorów.

Klasyfikator SVM - maszyna wektorów wspierających

Zakłada się w nim, że próbki uczące mają skończony charakter, tzn. w ogólności nie są reprezentatywne dla całości rozkładu. Klasyfikacja *k* klas ma charakter wielokrotnej klasyfikacji binarnej. Klasyfikator SVM jest optymalny z punktu widzenia kryterium zdolności do uogólniania dla nieistniejących próbek.

Klasyfikator Bayesa

Zakładamy, że rozkłady prawdopodobieństwa klas nad przestrzenią cech mają znaną postać (np. rozkład normalny) albo mogą być w pełni określone. Zadaniem konstrukcji (procesu uczenia) jest jedynie określenie parametrów tego rozkładu lub całego rozkładu (o nieparametrycznej postaci) dla każdej z rozpatrywanych klas. Zakłada się, że próbki uczące mają charakter reprezentatywny dla całości rozkładu. Wtedy klasyfikator Bayesa jest optymalny z punktu widzenia minimalizacji błędu klasyfikacji. Klasyfikator statystyczny może dysponować parametryczną lub nie-parametryczną funkcją gęstości prawdopodobieństwa.

Jako przypadki szczególne klasyfikatora Bayesa rozpatruje się klasyfikator „największej wiarygodności" i klasyfikator geometryczny minimalnej odległości.

Klasyfikator neuronowy - wielowarstwowy perceptron

Warto zastosować sieć neuronową do aproksymacji funkcji decyzyjnych w sytuacji, gdy nie wymagamy jawnych postaci tych funkcji a znalezienie ich na drodze analitycznej jest żmudne lub niemożliwe. Klasyfikator neuronowy dysponuje wysoką zdolnością do uczenia się funkcji decyzyjnej.

11.4. Klasyfikator według funkcji potencjału

W tym podejściu zakładamy, że:

- dla każdej klasy istnieje parametryczna funkcja (tzw. funkcja potencjału) zdefiniowana nad przestrzenią cech, charakteryzująca stopień przynależności danego punktu przestrzeni do zadanej klasy;
- wszystkie funkcje potencjału należą do jednej zadanej rodziny parametrycznych funkcji.

Najprostszą postacią takiej funkcji jest liniowa funkcja parametrów a :

$$d(c,a) = \{\mathbf{a}^T \varphi(c) \mid a \in \Re_a, \varphi_i(c)(i = 1,\cdots,m)\}, \tag{11.8}$$

w której dodatkowym ułatwieniem może być przyjęcie liniowej zależności od wektora cech (hiperpłaszczyzny):

$$\varphi(c) = (1, c_1, c_2, \cdots, c_n)^T, \tag{11.9}$$

gdzie a jest wektorem o $(n+1)$ elementach.

Bardziej złożoną funkcją jest funkcja kwadratowa (wielomian rzędu 2) n zmiennych:

$$\varphi(c) = (1, c_1, c_2, \cdots, c_n, c_1c_1, c_2c_1, \cdots, c_nc_n)^T \tag{11.10}$$

gdzie a jest wtedy wektorem $(1+n + n(n+1)/2)$-elementowym.

11.4.1. Liniowe funkcje potencjału

Decyzja dla 2 klas

Istnieją tylko dwie klasy, tzn. mamy tu binarny problem klasyfikacji. Definiujemy dla każdej z klas Ω_1 i Ω_2 liniowe funkcje $\mathrm{d}_1(\mathbf{c}, \mathbf{a}^{(1)})$ i $\mathrm{d}_2(\mathbf{c}, \mathbf{a}^{(2)})$, należące do jednej rodziny funkcji o parametrach,

$\mathbf{a} = [a_0, a_1, ..., a_n]$:

$$\mathrm{d}_1(\mathbf{c}, \mathbf{a}^{(1)}) = a_0^{(1)} + \sum_{i=1}^{n} a_i^{(1)} \cdot c_i, \tag{11.11}$$

$$d_2(\mathbf{c}, \mathbf{a}^{(2)}) = a_0^{(2)} + \sum_{i=1}^{n} a_i^{(2)} \cdot c_i$$

Kryterium decyzyjne wyznaczone jest przez funkcję:

$$d(c) = d_1(\mathbf{c}, \mathbf{a}^{(1)}) - d_2(\mathbf{c}, \mathbf{a}^{(2)}), \tag{11.12}$$

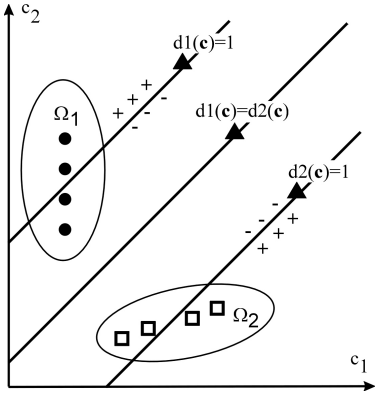
Reguła decyzyjna

Wybierz klasę Ω_1, gdy $d(\mathbf{c}) > 0$, tzn. gdy $d_1(c, a^{(1)}) > d_2(c, a^{(2)})$, lub wybierz klasę Ω_2 w przeciwnym razie; tzn.:

$$\zeta(c) = \begin{cases} \Omega_1, & gdy \quad d(c) > 0 \\ \Omega_2, & w\ przeciwnym\ razie \end{cases} \tag{11.13}$$

Klasyfikator mógłby też zwracać „brak decyzji" wtedy, gdy $d(\mathbf{c}) = 0$.

Przykład Binarna klasyfikacja dla 2-wymiarowych cech: funkcja decyzyjna niejawnie wyznacza prostą, dla której punktów zachodzi równość potencjałów obu klas (rys. 12.1). Ta linia prosta separuje 2 obszary klas w przestrzeni cech o współrzędnych $[c_1, c_2]$.



Rys. 11.6: Przykład liniowych funkcji potencjału dla problemu dwóch klas i 2-wymiarowego wektora cech.

Reguła decyzyjna dla k klas stanowi uogólnienie reguły podanej dla przypadku 2 klas i wynosi:

$$\varsigma(c) = \arg\max_k d_k(c, a^{(k)}) = \arg\max_k (a_0^{(k)} + \sum_{i=1}^{n} a_i^{(k)} \cdot c_i) \tag{11.14}$$

11.4.2. Uczenie się parametrów klasyfikatora

Niech $d_k(\mathbf{c}, \mathbf{a}^{(k)}) = 1$, gdy $\mathbf{c} \in \Omega_k$ (lub = -1, gdy $c \notin \Omega_k$). Dla każdej klasy o indeksie k istnieje N_k próbek uczących etykietowanych przynależnością do klasy. Dla każdej klasy można ułożyć układ równań o ($N = \sum N_k$) wierszach i (n+1) kolumnach (dla n+1 zmiennych tworzących wektor $\mathbf{a}^{(k)}$). Uzupełnijmy wektor $\mathbf{c}$ o zerową składową ($c_0 = 1$). Wektor oczekiwanych wyników $[-1 \text{ lub } 1]^T$ oznaczmy przez $\mathbf{d}_{(k)}$.

Ostatecznie dla każdej klasy k zapiszemy odrębny układ równań o postaci:

$$\mathbf{d}_{(k)} = \mathbf{C}\, \mathbf{a}^{(k)}, \tag{11.15}$$

Znalezienie $\mathbf{a}^{(k)}$ wymaga rozwiązania problemu liniowej optymalizacji metodą najmniejszych kwadratów MNK.

Optymalizacja liniowa MNK

Metoda **najmniejszych kwadratów** (MNK) jest podstawowym narzędziem optymalizacji. Wyjaśnimy jej istotę dla przypadku 2-wymiarowego. Wtedy celem metody MNK jest znalezienie liniowej zależności pomiędzy wielkościami b i a :

$$b = x\,a + y \ , \ \text{lub} \ x\,a + y \ - b = 0.$$

Na podstawie N obserwacji (pomiarów) punktów $(a_i\ ,\ b_i)$ mamy N równań :

$$\begin{aligned} & x\ a_1 + y \ - \mathrm{b}_1 = \varepsilon_1 \ . \\ & x\ a_2 + y \ - b_2 = \varepsilon_2 \ . \\ & \ldots\ldots\ldots\ldots\ldots\ldots \\ & x\ a_N + y \ - b_N = \varepsilon_N \ . \end{aligned} \tag{11.16}$$

Wprowadzamy wektor błędu aproksymacji: $\boldsymbol{\varepsilon} = [\ \varepsilon_1\ ,\ \varepsilon_2\ ,\ \ldots\ ,\ \varepsilon_N\ ]^T$.

Definiujemy cel optymalizacji – jest nim minimalizacja sumarycznego błędu kwadratowego:

$$\begin{aligned} U\,(x\ ,\ y) \ &= \varepsilon_1{}^2 \ + \varepsilon_2{}^2 \ + \ldots + \varepsilon_N{}^2 \ = \sum \varepsilon_i^2 \ = |\ \boldsymbol{\varepsilon}\ |^2 \\ U\,(x\ ,\ y\ \ ) \ &= \sum (x\ a_i + y - b_i)^2 \ . \end{aligned} \tag{11.17}$$

Minimum powyższej funkcji U wystąpi dla :

$$\frac{\partial U}{\partial x} = 0; \quad \frac{\partial U}{\partial y} = 0; \quad , \tag{11.18}$$

$$\frac{1}{2}\frac{\partial U}{\partial x} = \sum (x \cdot a_i + y - b_i) \cdot a_i = 0\ ; \qquad \frac{1}{2}\frac{\partial U}{\partial y} = \sum (x \cdot a_i + y - b_i) = 0 \tag{11.19}$$

Uzyskujemy 2 równania z 2 niewiadomymi:

$$\begin{aligned} x\ \sum a_i{}^2 + \ y\ \sum a_i \ &= \sum (a_i\, b_i) \\ x\ \sum a_i + \ y\, N \ &= \sum \ b_i \end{aligned} \tag{11.20}$$

Jeśli istnieje rozwiązanie układu (11.20) to przyjmuje ono postać:

$$\begin{bmatrix} x \\ y \end{bmatrix} = \frac{1}{N\sum a_i^2 - (\sum a_i)^2} \begin{bmatrix} N & -\sum a_i \\ -\sum a_i & \sum a_i^2 \end{bmatrix} \begin{bmatrix} \sum a_i b_i \\ \sum b_i \end{bmatrix} \tag{11.21}$$

11.5. Klasyfikator SVM

W klasyfikatorze statystycznym zakłada się, że dysponujemy reprezentatywną próbką uczącą dla wszystkich klas. W *maszynie wektorów podpierających* SVM (ang. „*Support Vector Machine*") przyjmuje się skończony charakter próbek uczących i sprowadza problem do wielokrotnej decyzji pomiędzy dwiema klasami (lub grupami klas) (oznaczanymi zwykle jako „+1", „-1").

Podczas procesu uczenia maszyny poszukuje się optymalnej hiperpłaszczyzny (dla przypadku liniowego) lub hiperpowierzchni (dla „nieliniowej" maszyny) separującej obszary cech dla różnych klas. Kryteria wpływające na proces uczenia to minimalizacja tzw. błędu empirycznego i rozmiar Vapnika-Czervonenkisa.

Rozmiar Vapnika-Czervonenkisa (rozmiar VC) h jest miarą dla zbioru funkcji rozdzielających. Dla problemu dwóch klas h oznacza maksymalną liczbę punktów w przestrzeni cech, które mogą zostać rozdzielone we wszystkie możliwe sposoby – liczba takich podziałów wynosi 2^h. Wystarczy przy tym, aby dla każdego sposobu podziału istniał przynajmniej jeden zbiór punktów o mocy h, tzn. istnieje wtedy przynajmniej jedna prawidłowa funkcja rozdzielająca. Specyficznym zbiorem funkcji rozdzielających jest zbiór zorientowanych hiperpłaszczyzn:

$$d_{\tilde{\mathbf{a}}}(\mathbf{c}) = \mathbf{c}^T\mathbf{a} + a_0, \quad \tilde{a} = \begin{pmatrix} a_0 \\ \mathbf{a} \end{pmatrix} \tag{11.22}$$

Za ich pomocą określamy to, czy wektor cech $\mathbf{c}$ leży po „dodatniej" stronie lub po „ujemnej" stronie płaszczyzny czy też na samej płaszczyźnie rozdzielającej. Można pokazać, że rozmiar Vapnika-Chervonenkisa zbioru zorientowanych hiperpłaszczyzn w n-wymiarowej przestrzeni cech $\Re^n$ wynosi: $h = n + 1$.

Przykład. Na płaszczyźnie zbiór trzech punktów daje się rozdzielić za pomocą zorientowanej prostej na ($2^3 = 8$) sposobów (rys. 11.7). To nie zachodzi już dla czterech punktów, czyli dla płaszczyzny (n=2): h=3.

(a) (b)

Rys. 11.7: Ilustracja rozdziału zbioru punktów na płaszczyźnie: (a) 4 partycje uzyskane dzięki jednej zorientowanej prostej, pozostałe 4 powstają po zmianie zorientowania prostej; (b) podział 4 punktów jedną prostą w ogólności nie jest możliwy.

11.5.1. SVM dla liniowo separowalnego zbioru próbek

Zakładamy istnienie próbek dwóch klas liniowo separowalnych. Niech istnieje *N* próbek uczących zaklasyfikowanych jako „dodatnie próbki" ($y_i = 1$) lub „negatywne próbki" ($y_i = -1$). Jeśli istnieje hiperpłaszczyzna o parametrach $\mathrm{a} = [a_0, a_1, \ldots, a_n]^T$, która rozdziela próbki uczące to zachodzi:

$$\begin{aligned} {}^j\mathbf{c}^T\mathbf{a} + a_0 &\geq +1, \quad if \quad y_j = +1 \\ {}^j\mathbf{c}^T\mathbf{a} + a_0 &\leq -1, \quad if \quad y_j = -1 \\ y_j({}^j\mathbf{c}^T\mathbf{a} + a_0) &\geq +1, \quad \forall\, {}^j\mathbf{c} \in \omega \end{aligned} \tag{11.23}$$

Dla pierwszej hiperpłaszczyzny w (11.23) zachodzą zależności:

$$\begin{aligned} \mathbf{n} &= \frac{-\mathbf{a}}{\sqrt{\mathbf{a}^T\mathbf{a}}} = \frac{-\mathbf{a}}{|\mathbf{a}|}, && \text{Jednostkowy wektor normalny} \\ s_0 &= \frac{-a_0}{|\mathbf{a}|}, && \text{Odległość od początku układu} \\ s_{\mathbf{c}} &= \frac{-(\mathbf{a}^T\mathbf{c} + a_0)}{|a|}, && \text{Odległość od punktu } c \end{aligned} \tag{11.24}$$

„Pozytywna" odległość punktu od hiperpłaszczyzny zachodzi wtedy, gdy punkt leży w kierunku wektora normalnego do powierzchni. Płaszczyzny w zależnościach (11.23) są znormalizowane tak, że dla punktu c' najbliższego płaszczyźnie zachodzi: $|\mathbf{c}'^T\mathbf{a} + a_o| = 1$.

Rozpatrzmy punkty, które leżą na hiperpłaszczyznach

$(\mathbf{c}^T\mathbf{a} + a_o = 1)$ lub $(\mathbf{c}^T\mathbf{a} + a_o = -1)$.

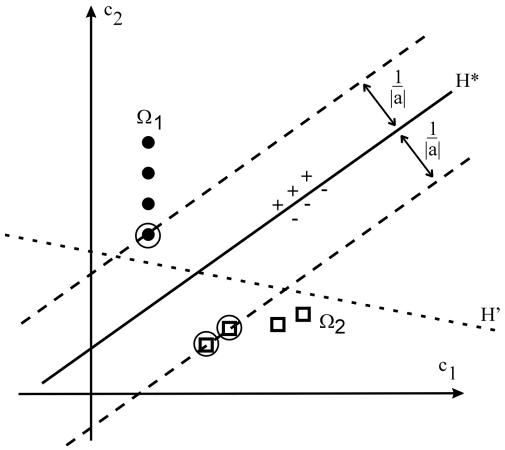
Obie hiperpłaszczyzny posiadają ten sam wektor normalny $\mathbf{a}$, tzn. są równoległe do siebie. Ich odległości od środka układu wynoszą odpowiednio:

$|1 - a_o| / |\mathbf{a}|$ i $|-1 - a_o| / |\mathbf{a}|$,

czyli ich wzajemny odstęp wynosi $2 / |\mathbf{a}|$. Pomiędzy tymi dwiema hiperpłaszczyznami nie mogą leżeć żadne próbki uczące.

Sensownym kryterium podlegającym optymalizacji jest maksymalizacja odstępu obu hiperpłaszczyzn czyli minimalizacja $|\mathbf{a}|^2$. Uzyskaną w ten sposób hiperpłaszczyznę optymalną oznaczmy przez H* (rys. 11.8).

Te z próbek uczących, które leżą dokładnie na hiperpłaszczyznach związanych z H*, tzn. dla których zachodzi równość w równaniach (11.23) nazywamy „wektorami wspierającymi". Są one wystarczające i konieczne do tego, aby wyznaczyć hiperpłaszczyznę optymalną – pozostałe próbki uczące mogą być pominięte bez szkody dla wyznaczenia H*. Z drugiej strony pominięcie jakiegoś wektora wspierającego zmieniłoby rozwiązanie dla H*.



Rys. 11.8: Optymalna płaszczyzna rozdzielająca H* dla próbek dwóch klas. Mogą istnieć inne płaszczyzny rozdzielające (np. H') ale nie są one optymalne w sensie przyjętych kryteriów.

Optymalny liniowy klasyfikator dla liniowo separowalnego zbioru próbek należących do 2 klas wyznaczony jest przez hiperpłaszczyznę:

$$H^*: d_{\tilde{\mathbf{a}}}(\mathbf{c}) = \mathbf{c}^T\mathbf{a} + a_0 = 0 \tag{11.25}$$

uzyskaną dzięki minimalizacji funkcji celu (kryterium)

$$\min_{\tilde{a}\in R^n} f(\tilde{\mathbf{a}}) = \min_{\tilde{a}\in R^n}(\frac{1}{2}|\,\tilde{\mathbf{a}}\,|^2) \tag{11.26}$$

przy N warunkach dodatkowych , $C_i(\mathrm{a}) \geq 0$, i =1, ..., N :

$$y_j({}^j\mathbf{c}^T\mathbf{a} + a_0) - 1 \geq 0, \quad \forall\, {}^j\mathbf{c} \in \Omega \tag{11.27}$$

Reguła decyzyjna jest postaci:

$$\zeta(c) = \begin{cases} \Omega_1, & gdy\ \ d_{\tilde{\mathbf{a}}}(c) \geq 0 \\ \Omega_2, & gdy\ \ d_{\tilde{\mathbf{a}}}(c) < 0 \end{cases} \tag{11.28}$$

Znalezienie rozwiązania problemu (11.27-12.27) wymaga zastosowania metody **wypukłego programowania kwadratowego** z nierównościowymi ograniczeniami liniowymi.

Wektory wspierające

Okazuje się, że równanie szukanej hiperpłaszczyzny możemy wyrazić niezależnie od wektora parametrów $\mathbf{a}$ w postaci:

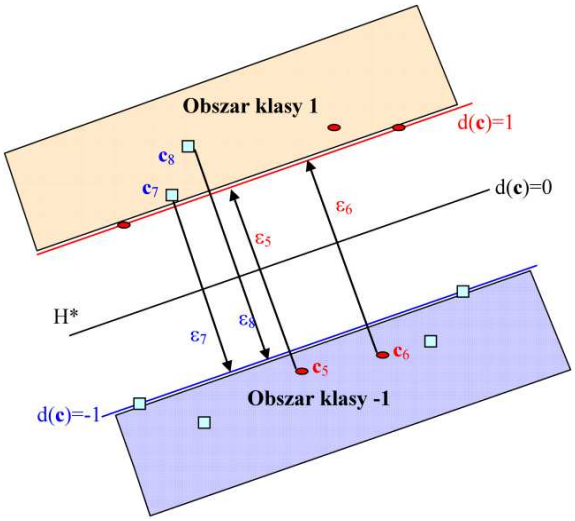
$$H^*: d_{\tilde{\mathbf{a}}}(\mathbf{c}) = \sum_{j=1}^{N} \vartheta_j y_j({}^{T\ j}\mathbf{c}) + a_0 = 0 \tag{11.29}$$

gdzie ϑ_j są nieujemnymi tzw. mnożnikami Lagrange'a, pozwalającymi ważyć wpływ poszczególnych wektorów cech. Równanie (11.29) wyraża znaną nam już obserwację, że do wyznaczenia

hiperpłaszczyzny wystarczą nam produkty skalarne wybranych wektorów cech. Ta obserwacja pozwoli nam na łatwe uogólnienie liniowej maszyny SVM do nieliniowej SVM.

Dla próbek (wektorów cech) leżących dokładnie na hiperpłaszczyźnie mnożniki wynoszą: $\vartheta_j=0$ lub $\vartheta_j>0$, a dla próbek nie leżących na hiperpłaszczyźnie, mnożniki są zerowane: $\vartheta_j = 0$). Do obliczenia optymalnej hiperpłaszczyzny wchodzą tylko wektory cech o aktywnych warunkach dodatkowych ($\vartheta_j>0$) **– wektory wspierające** - wszystkie pozostałe wektory są zbędne.

11.5.2. Liniowa SVM dla liniowo nieseparowalnych próbek



Rys. 11.9: Dodajemy element "kary" do funkcji celu dla każdej błędnie zaklasyfikowanej próbki uczącej. W powyższym przypadku proporcjonalnie do odległości ε_5 i ε_6 (dla próbek klasy 1) i ε_7 i ε_8 (dla próbek klasy -1).

Jeśli zbiór próbek nie jest w pełni liniowo separowalny, ale błędne próbki zdarzają się sporadycznie, to możemy je traktować jako przypadkowe błędy i nadal stosować liniową SVM (rys. 11.9). Należy jednak wtedy rozszerzyć funkcję celu o dodatkową składową błędu:

$$\min_{\tilde{a}\in R^n} f(\tilde{\mathbf{a}}) = \min_{\tilde{a}\in R^n}\left(\frac{1}{2}\,|\,\tilde{\mathbf{a}}\,|^2 + C\sum_{j=1}^{N}\varepsilon_j\right) \tag{11.30}$$

gdzie C jest ustalonym parametrem, ważącym wpływ szumu na łączną funkcję celu, a wartości ε_j to odległości błędnych próbek od hiperpłaszczyzn dla właściwej klasy. Ograniczenia dodatkowe przyjmują wtedy postać:

$$y_j({}^j\mathbf{c}^T\mathbf{a} + a_0) \geq 1 - \varepsilon_j, \quad \forall\, {}^j\mathbf{c} \in \omega, \varepsilon_j \geq 0 \tag{11.31}$$

Oczywiście, zamiast $n+1$ zmiennych, mamy teraz $n+1+N$ zmiennych. Rozwiązanie tego problemu ma identyczną postać, co równanie (11.29), zmienia się jedynie zakres wartości dla mnożników Lagrange'a i wynosi on:

$$\begin{aligned} 0 \leq \vartheta_j \leq C \\ 0 = \sum_{j=1}^{N} \vartheta_j\, y_j \end{aligned} \tag{11.32}$$

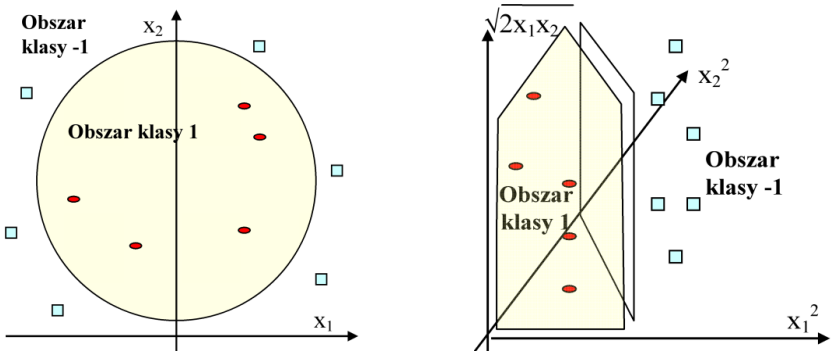
11.5.3. Nieliniowe maszyny SVM

Zwykle nie oczekujemy od zbioru próbek zadanych dla złożonych rzeczywistych problemów, aby można było zastosować dla nich liniowy SVM. Jednak możemy dokonać linearyzacji cech prze-

kształcając przestrzeń cech w wyżej wymiarową przestrzeń $F(\mathrm{c})$. W tym przekształceniu zastępujemy wektory c i ${}^{j}\mathrm{c}$ podane w równaniu (11.29) przez $F(\mathrm{c})$ i $F({}^{j}\mathrm{c})$:

$$H^*: d_{\tilde{\mathbf{a}}}(\mathbf{c}) = \sum_{j=1}^{N} \vartheta_j y_j (F(\mathbf{c})^T F({}^{j}\mathbf{c})) + a_0 = 0 \tag{11.33}$$

Często produkt skalarny wektorów, $F(\mathrm{c})^T F({}^{j}\mathrm{c})$, może być obliczony analitycznie co oznacza, że nie będzie potrzeby obliczania $F(\ )$ dla każdej próbki z osobna. Np. jeśli ($F_1 = x_1{}^2,\ F_2 = x_2{}^2, F_3 = \sqrt{2}x_1x_2$) to okazuje się, że: $F(\ \mathrm{c}\ )^T F({}^{j}\mathrm{c}\ ) = (\mathrm{c} \cdot {}^{j}\mathrm{c})^2$ (rys. 11.10).



(a) (b)

Rys. 11.10: (a) Liniowo nieseparowalny zbiór 2-wymiarowych próbek uczących (próbki klasy +1 zaznaczono jako wypełnione, a próbki klasy -1 jako okręgi). Szukana (nieliniowa) funkcja decyzyjna jest postaci: $x_1{}^2 + x_2{}^2 \le 1$. (b) Po odwzorowaniu tych danych do przestrzeni 3-wymiarowej: ($x_1{}^2, x_2{}^2, \sqrt{2}x_1x_2$), następuje linearyzacja funkcji decyzyjnej.

Takie wyrażenie nazywane jest funkcją "jądra" tego przekształcenia (ang. *kernel function*). Przykłady funkcji jądra to :

- wielomian: $K(\mathrm{x}, \mathrm{y}) = (\mathrm{x}^{\mathrm{T}}\ \mathrm{y} + 1)$
- funkcja radialna: $K(\mathbf{x}, \mathbf{y}) = \exp\left(-\frac{(\mathbf{x}-\mathbf{y})^2}{2\sigma^2}\right)$
- postać neuronowa: $K(\mathrm{x}, \mathrm{y}) = \tanh(\kappa\ \mathrm{x}^{\mathrm{T}}\ \mathrm{y} - \delta)$

11.6. Pytania

1. Na czym polega zadanie uczenia klasyfikatora stosującego drzewo decyzyjne?
2. Omówić algorytm uczenia DTL.
3. Na czym polega stosowanie entropii dla wyboru testu podczas uczenia DTL?
4. Omówić funkcję decyzyjną klasyfikatora wg funkcji potencjału.
5. Omówić problem optymalizacji MNK.
6. Na czym polega liniowy klasyfikator SVM? Co to są wektory wspierające?
7. Na czym polega i jak rozwiązujemy problem optymalizacji kwadratowej z nierównościowymi ograniczeniami liniowymi?
8. Jak definiujemy nieliniowy klasyfikator SVM?

11.7. Zadania

Zad. 11.1

Wykonać uczenie drzewa decyzyjnego metodą DTL, w którym wybór testu (atrybutu) dokonuje się na podstawie przyrostu informacji (maksymalizacji entropii), dla podanego zbioru uczącego (przeprowadzić niezbędne obliczenia).

Tab. 11-2

Próbka X	Atrybut a1	Atrybut a2	Atrybut a3	Klasa c
1	1	1	1	0
2	1	1	2	0
3	2	1	3	0
4	2	2	2	0
5	2	2	3	0
6	1	1	3	1
7	1	2	2	1
8	1	2	3	1
9	2	1	1	1
10	2	2	1	1

Zad. 11.2

Zdefiniować i rozwiązać problem optymalizacyjny (poprzez docelowe układy równań) występujący podczas projektowania (uczenia) klasyfikatora według liniowych funkcji potencjału; dla 2-wymiarowego wektora cech i dwóch klas. Zadany jest zbiór próbek uczących (tab. 11-3).

Tab. 11-3

I	1	2	3	4	5	6
$C^i = (c^i_1, c^i_2)$	(1; 2)	(1; 3)	(1; 4)	(1; 1)	(2; 1)	(3; 1)
$\Omega_k =$	Ω_1	Ω_1	Ω_1	Ω_2	Ω_2	Ω_2

Zad. 11.3

Przeanalizować możliwe rozwiązania problemu projektowania (uczenia) klasyfikatora - liniowej maszyny SVM, dla 2-wymiarowego wektora cech i dwóch klas, gdy zadany jest zbiór próbek uczących jak w tabeli 11.4:

- Podać spodziewane w tym przypadku rozwiązanie w graficzny sposób,
- Rozwiązać problem analitycznie lub algorytmicznie i podać dokładne rozwiązanie.

Tab. 11.4

i	1	2	3	4	5	6
$C^i = (c^i_1, c^i_2)$	(1; 2)	(1; 3)	(1; 4)	(1; 1)	(2; 1)	(3; 1)
$\Omega_k =$	Ω_1	Ω_1	Ω_1	Ω_2	Ω_2	Ω_2

12. Uczenie się aproksymacji

12.1. Zadanie aproksymacji funkcji

Uczenie się aproksymacji funkcji jest kolejną, po uczeniu się klasyfikacji i uczeniu się pojęć, odmianą uczenia indukcyjnego.

W zadaniu aproksymacji poszukujemy reprezentacji nieznanej funkcji,

$$f(\mathbf{x})\colon \mathbf{X} \to \Re,$$

o wartościach rzeczywistych. Wyznaczyć należy taką funkcję,

$$h(\mathbf{x};\, \mathbf{p}),\ \text{z parametrami}\ \mathbf{p} = [p_1\ , p_2\ , \ldots\ , p_n]^{\mathrm{T}},$$

której wartości w badanej dziedzinie $\mathbf{X}$ dostatecznie mało różnią się, w sensie przyjętego kryterium, od odpowiednich wartości funkcji $f(\mathbf{x})$.

Informacja trenująca - próbka ucząca to para $(\mathbf{x}, f(\mathbf{x}))$ dla $\mathbf{x}\in\mathbf{X}$.

Kryterium oceny aproksymacji

Dla większości algorytmów jako kryterium oceny wygodne jest stosowanie błędu średniokwadratowego zdefiniowanego jako:

$$e_p = \frac{1}{N}\sum_{x_i \in \mathrm{T}} (f(\mathbf{x}_i) - h(\mathbf{x}_i, \mathbf{p}))^2$$

gdzie N oznacza liczbę próbek uczących w zbiorze treningowym $\mathbf{T}$.

Rola aproksymowanej funkcji

Często tak aproksymowana funkcja pełni następnie rolę funkcji decyzyjnej lub funkcji potencjału w zagadnieniach klasyfikacji i charakteryzuje ona stopień przynależności danej próbki do zadanej klasy.

12.2. Metodyka uczenia się aproksymacji funkcji

Niech f oznacza nieznaną nam docelową funkcję.

Znane są **przykłady** (obserwacje) tej funkcji dla zadanego wejścia x – czyli pojedyncza obserwacja (**próbka ucząca**) to para $(x, f(x))$.

Zadanie aproksymacji w uczeniu przez indukcję jest:

mając dany zbiór próbek uczących znaleźć funkcję **(hipotezę)** *h,* aproksymującą nieznaną funkcję, tzn. $h \approx f$.

Jak zmierzyć jakość hipotezy, tzn. bliskość hipotezy i rzeczywistej funkcji, tzn. czy $h \approx f$?

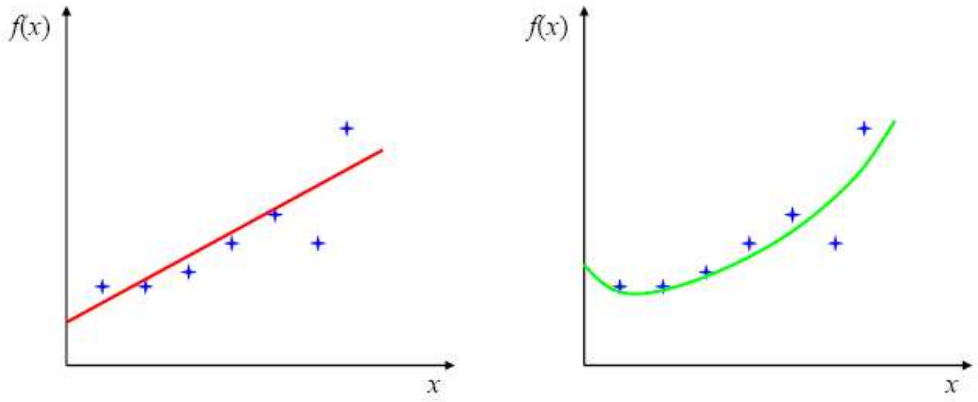
Ocena jakości hipotezy

Jakość hipotezy h można wyrazić poprzez ocenę jej zdolności do generalizacji, tzn. tego, jak dobrze przewiduje ona wartości funkcji f dla nieznanych dotąd obserwacji.

Wykonując testy zgodności hipotez uzyskanych dzięki uczeniu na zbiorach próbek uczących o różnych rozmiarach możemy przedstawić jakość hipotez w postaci krzywej zgodności hipotez (rys. 12.1): wykres przedstawia procentową zgodność hipotezy z próbkami w zbiorze testowym jako funkcję rozmiaru zbioru próbek.

Rys. 12.1: Krzywa zgodności hipotez

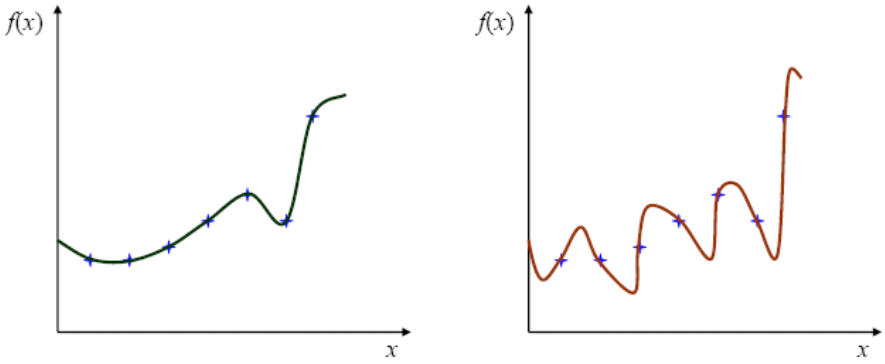
Załóżmy, że znana jest aproksymacja zbioru punktów na płaszczyźnie linią prostą lub krzywą (funkcja 1-wymiarowa) (rys. 12.2).



Rys. 12.2: Aproksymacja zbioru próbek funkcja liniową i kwadratową

Hipoteza jest **zgodna z próbkami** uczącymi, jeśli wszystkie je spełnia. Powyższe hipotezy (rys. 12.2) nie są zgodne z próbkami. Dlatego kontynuujemy szukanie hipotezy o postaci wielomianu wyższego rzędu.

Może być wiele funkcji zgodnych z próbkami uczącymi. Kierujemy się wtedy zasadą wyboru funkcji najprostszej spośród zgodnych funkcji (rys. 12.3).



Rys. 12.3: Dwie funkcje wielomianowe zgodne z próbkami uczącymi. Lewa funkcja jest prostsza i powinna być wybrana jako hipoteza.

12.3. Funkcja parametryczna

W klasycznym zadaniu **aproksymacji parametrycznej** poszukiwana funkcja ma postać parametryczną, zdefiniowaną nad przestrzenią cech jako:

$$h(\boldsymbol{x}, \boldsymbol{p}) = \boldsymbol{p}^T \varphi(\boldsymbol{x}), \quad \boldsymbol{p} \in \Re^m, \varphi(\boldsymbol{x}) \in \Re^m$$

Każdy przykład (próbka ucząca), x ∈ X , jest reprezentowany przez wektor cech, $c = \varphi(x)$, gdzie $\varphi = [\varphi_0, \ldots, \varphi_m]^T$ jest wektorem funkcji wyznaczających wartości cech.

Dla każdej klasy funkcja $h()$ ma tę samą postać, a różni się jedynie unikalnym wektorem wartości parametrów **p**. W istocie jest to funkcja liniowa nad przestrzenią cech, gdzie jednak tzw. **funkcje bazowe**, $\varphi_i(x)$, są potencjalnie nieliniowe.

Zadania w aproksymacji funkcji parametrycznej:

1. poszukiwanie zestawu funkcji bazowych $\varphi(x)$,
2. minimalizacja błędu e_p względem wektora parametrów **p**.

12.3.1. Funkcja liniowa

Liniowa funkcja bazowa przedstawia bezpośrednią zależność wektora cech od próbki:

$$\boldsymbol{c} = \varphi(\boldsymbol{x}) = [1, x_1, x_2, \cdots, x_m]^T$$

gdzie **c** jest wektorem o $(m+1)$ elementach.

Przykład nieliniowej funkcji bazowej to **funkcja kwadratowa** (wielomian rzędu 2) o m zmiennych (składowych **x**):

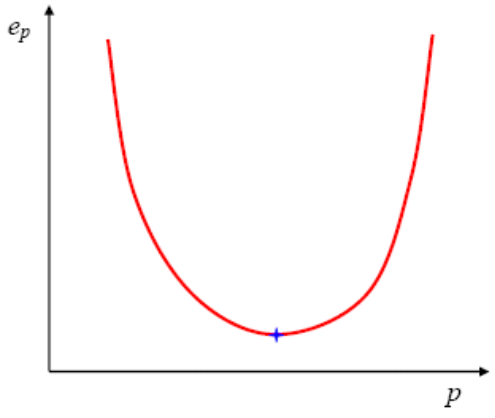
$$\varphi(\boldsymbol{x}) = [1, x_1, x_2, \cdots, x_n, x_1x_1, x_2x_1, \cdots, x_nx_n]^T$$

Teraz **c** będzie wektorem $(1+m + m(m+1)/2)$-elementowym i tyle też potrzebnych będzie parametrów w wektorze **p**.

Nieliniowe funkcje bazowe wyższego rzędu w ogólności również mogą zależeć od zbioru parametrów, tak jak poszukiwana aproksymacja funkcji $f(\mathbf{x})$.

Błąd średniokwadratowy e_p to funkcja kwadratowa, więc dla zmiennej x o jednym wymiarze i **liniowych funkcji** f, h ma ona jedno rozwiązanie i osiąga minimum w miejscu zerowania się pochodnej.

Mamy więc w punkcie rozwiązania: $\dfrac{\mathrm{d}\, e_p}{\mathrm{d}\, p} = 0$



Rys. 12.4: Minimum funkcji kwadratowej.

Dla uczenia aproksymacji **liniowej** funkcji **wielowymiarowej** *f* poszczególne wymiary potraktujemy jako niezależne i rozwiążemy układ *m* równań o m niewiadomych p_i :

$$\frac{\mathrm{d}\, e_p}{\mathrm{d}\, p_i} = 0\,,\; i = 0,1,...,m$$

W przypadku funkcji liniowych znalezione rozwiązanie jest minimum globalnym funkcji błędu.

Przykład liniowej funkcji parametrycznej

Załóżmy rodzinę liniowych funkcji o parametrach $\mathbf{p}$, $h(\mathbf{x}, \mathbf{p})$, gdzie $\mathbf{p} = [p_0, p_1, ..., p_n]$

Czyli:

$$h(\boldsymbol{x}, \boldsymbol{p}) = p_0 + \sum_{i=1}^{m} p_i \cdot x_i$$

Przykłady trenujące nie musza leżeć na jednej prostej (płaszczyźnie, hiperpłaszczyźnie) w przestrzeni **X** , ale próbujemy jednak aproksymować je funkcją liniową.

Niech danych jest *N* próbek uczących w zbiorze **T**. Można ułożyć układ równań o (*N*) wierszach i (*m*+1) kolumnach (dla *m*+1 parametrów w wektorze $\mathbf{p}$).

Uzupełniamy każdą próbkę $\mathbf{x}$ o zerową składową $(x_0 = 1)$ i otrzymujemy macierz współczynników próbek

$$\boldsymbol{T} = \begin{pmatrix} 1 & \boldsymbol{x}_1^T \\ 1 & \boldsymbol{x}_2^T \\ \vdots & \vdots \\ 1 & \boldsymbol{x}_{N-1}^T \\ 1 & \boldsymbol{x}_N^T \end{pmatrix} = \begin{pmatrix} 1 & x_{1,1} & x_{1,2} & \cdots & x_{1,m} \\ 1 & x_{21} & x_{2,2} & \cdots & x_{2,m} \\ \vdots & \vdots & \ddots & & \vdots \\ 1 & x_{N-1,1} & x_{N-1,2} & \cdots & x_{N-1,m} \\ 1 & x_{N,1} & x_{N,2} & \cdots & x_{N,m} \end{pmatrix}$$

Wektor oczekiwanych wyników funkcji $h(\mathbf{x})$ dla próbek uczących oznaczmy przez $\mathbf{d}$.

Wektor błędów składowych (liczony po parametrach) dla aproksymacji funkcji wyznaczamy jako:

$$\boldsymbol{\varepsilon} = \mathbf{d} - \mathbf{T}\,\mathbf{p}$$

Średni błąd kwadratowy wynosi:

$$\| e \|^2 = \frac{1}{N} \sum_{i=1}^{N} \varepsilon_i^2$$

Teraz należy znaleźć minimum powyższej funkcji błędu, stosując metodę najmniejszych kwadratów (MNK), przyrównując pochodne cząstkowe do zera i rozwiązując tak wyznaczony układ równań.

12.3.2. Reguła poszukiwania wzdłuż ujemnego gradientu

Jeśli funkcje f, h **nie są liniowe** to poprzednie równania różniczkowe nie prowadzą do układu równań liniowych. Wtedy możemy zastosować metodę iteracyjnego poprawiania poprzedniego oszacowania (uzyskanego w *k*-tej iteracji) $\mathbf{p}(\mathrm{k})$ przesuwając się wzdłuż kierunku ujemnego gradientu funkcji błędu, danego w przestrzeni parametrów. Jest to tzw. **uogólniona reguła delta**:

$$\boldsymbol{p}(k+1) = \boldsymbol{p}(k) + \Delta \boldsymbol{p} \quad ,$$

gdzie

$$\Delta \boldsymbol{p} = -\beta \cdot \nabla_p e_p$$

jest przeskalowanym wektorem pochodnych cząstkowych:

$$\nabla_p e_p = [\frac{\partial e_p}{\partial p_0}, \frac{\partial e_p}{\partial p_1}, \cdots, \frac{\partial e_p}{\partial p_m}]$$

a współczynnik skalujący, $\beta \in (0,1]$.

W *k*-tej iteracji wyznaczamy wektor modyfikacji $\Delta\mathbf{p}$ jako:

$$\Delta\mathbf{p} = \beta \frac{2}{N} \sum_{x_i \in \mathrm{T}} (f(\mathbf{x}_i) - h(\mathbf{x}_i, \mathbf{p}(k))) \cdot \nabla_p h(x_i, p(k))$$

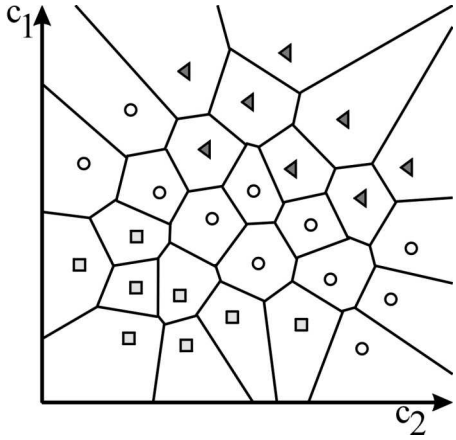
Posługiwanie się regułą spadku gradientu dla funkcji nieliniowych h nie daje gwarancji, że znalezione minimum dla funkcji błędu jest jednocześnie minimum globalnym tej funkcji, gdyż osiągnięte minimum zależy od przyjętej wartości początkowej parametrów $\mathbf{p}$.

12.4. Model pamięciowy aproksymacji funkcji

Przedstawimy teraz zupełnie odmienne podejście do uczenia się aproksymacji funkcji. Tzw. pamięciowa metoda uczenia się nie przetwarza przykładów trenujących a tylko te przykłady zapamiętuje.

Załóżmy istnienie zbioru próbek uczących, $\mathrm{T} = \{\mathrm{c}_1, \mathrm{c}_2, ..., \mathrm{c}_N\}$, dla których znane są wartości nieznanej funkcji $f(\mathrm{c}_i)$.

W pamięciowym modelu uczenia każda próbka zostaje zapamiętana, staje się reprezentantem funkcji a wartość jej etykiety stanowi aproksymację wartości funkcji dla „*zajmowanego przez nią*" obszaru w przestrzeni reprezentacji. Zbiór punktów przestrzeni cech, dla których najbliższym sąsiadem spośród istniejących cech jest c_i nazywamy *komórką Voronoia* dla c_i. Zbiór przykładów uczących T wyznacza *podział Voronoia* całej przestrzeni cech (rys. 12.5).



Rys. 12.5: Ilustracja podziału Voronoia dla 2-wymarowych cech.

Aproksymacja według *najbliższego sąsiada*

Aproksymuj wartość funkcji w punkcie $\boldsymbol{x}$ przez wartość pamiętanej próbki $\boldsymbol{c}$ położonej najbliżej zadanej wartości $\boldsymbol{x}$.

$$h(\boldsymbol{x}) = f(\boldsymbol{c})$$

gdzie $\mathbf{c}_{opt} = \arg\min_{\mathbf{c}_i \in \mathrm{T}} \|\mathbf{x} - \mathbf{c}_i\|$

Aproksymacja według *k sąsiadów*

Aproksymuj wartość funkcji w punkcie x przez średnią wartość k pamiętanych próbek c_i położonych najbliżej zadanemu punktowi x spośród próbek zbioru T:

$$h(\boldsymbol{x}) = \frac{1}{k}\sum_{i=1}^{k} f(\boldsymbol{c}_i)$$

12.5. Pytania

1. Na czym polega zadanie aproksymacji parametrycznej?
2. Jak wyznaczamy parametry aproksymacji funkcji?
3. Wyjaśnić tryb uczenia funkcji potencjału dla liniowego klasyfikatora.
4. Omówić model pamięciowy w uczeniu się aproksymacji funkcji.

12.6. Zadania

Zad. 12.1

Stosując uogólnioną regułę delty zdefiniować układy równań a następnie je rozwiązać, w celu wyznaczenia parametrów aproksymatorów liniowych funkcji potencjału dla binarnego klasyfikatora.

Zadany jest zbiór próbek uczących:

i	1	2	3	4	5	6
$C^i = (c^i_1, c^i_2)$	(1; 2)	(1; 3)	(1; 4)	(1; 1)	(2; 1)	(3; 1)
$\Omega_k =$	Ω_1	Ω_1	Ω_1	Ω_2	Ω_2	Ω_2

Zad. 12.2

Informacja o nieliniowej zależności dwóch zmiennych a, b dana jest w postaci wyników $N=9$ niezależnych pomiarów obarczonych błędami przypadkowymi:

a = [0, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0]

b = [1.02, 0.88, 0.48, 0.44, 0.33, 0.30, 0.15, 0.14, 0.12]

Dokonać aproksymacji funkcji, b = f(a), wielomianem:

a) drugiego, i

b) trzeciego stopnia.

Zad. 12.3

Zastosować model pamięciowy uczenia się aproksymacji funkcji nad przestrzenią cech $\Re^2$. Zadanych jest 11 próbek uczących:

c= (x, y)	(1;2,5)	(1; 3)	(1;3,5)	(2; 3)	(4; 3)	(2; 2)	(1; 1)	(2; 0)	(3; 1)	(3,5;1,5)	(4;1,5)
f(**c**)	1	1	1	2	2	3	3	3	3	4	4

Podać wartości aproksymowanej funkcji w metodzie 4 najbliższych sąsiadów dla cech (2; 2,5), (3, 3) i (1, 2).

Zad. 12.4

Zilustrować zasadę modelu pamięciowego w uczeniu się aproksymacji funkcji, z zastosowaniem algorytmu k sąsiadów (kNN), dla *k* = 3. Niech pamięć zawiera przykłady trenujące T= { x_1,x_2,...,x_{25}}, reprezentowane pojedynczą cechą *c* o postaci:

$$c_i = \varphi(x_i) = 0.006\pi \cdot i^{3/2}$$

Wartości przyjmowane dla tych przykładów przez funkcję docelową wynoszą:

$$f(c) = \frac{\sin c}{c}$$

Obliczyć błąd względny i błąd średniokwadratowy dla hipotezy funkcji reprezentowanej przez zapamiętane przykłady na zbiorze przykładów P=(x_1',x_2',...,x_8'), zadanych jako:

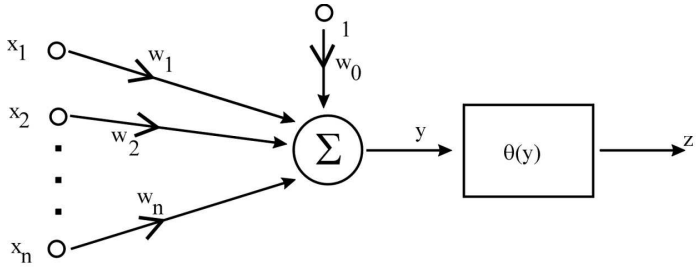
$$\varphi(x_i') = 0.05\pi \cdot i^{4/3}$$

13. Uczenie sieci neuronowej MLP

13.1. Model neuronu

Sztuczne sieci neuronowe są dogodnym narzędziem stosowanym tam, gdzie podanie rozwiązania o analitycznej postaci funkcji jest trudne lub niemożliwe. Dzięki algorytmom uczenia sieci na podstawie zadanego zbioru próbek możemy wyznaczyć sieć neuronową dobrze *aproksymującą* nieznaną postać funkcji.

13.1.1. Struktura neuronu



Rys. 13.1 Model neuronu (Rosenblatt, 1957)

Wejścia: $(x_1, x_2, \ldots, x_n)$.

Wyjście: z.

Funkcja pobudzenia (wyjściowa): $$y = \sum_{i=1}^{n} (w_i \cdot x_i) - w_0 \cdot 1$$

gdzie w_0 jest wagą dodatkowego wejścia progowego.

Funkcja aktywacji: $z = \theta(y)$.

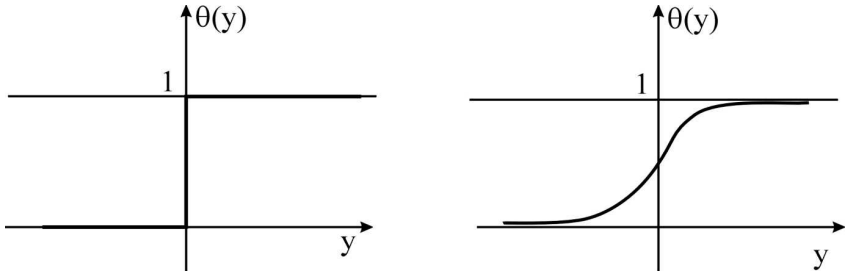
13.1.2. Funkcja aktywacji neuronu

Typowe funkcje aktywacji to:

1. Funkcja liniowa, $z = k\,y$, gdzie k jest zadanym stałym współczynnikiem.
2. Funkcja skoku jednostkowego: $z = \begin{cases} 1 & \text{gdy } y_i > y_T \\ 0 & \text{, przeciwnie} \end{cases}$

   gdzie y_T jest zadanym progiem.
3. Funkcja *logistyczna* (*sigmoidalna unipolar.*): $\theta(y) = \dfrac{1}{1+\exp(-\beta y)}$, gdzie β jest zadanym parametrem. Wyjście neuronu przyjmuje wartości z przedziału [0, 1].
4. Funkcja *tangens hiperboliczny* (*bipolarna*): $\theta(y) = \mathrm{tgh}(\dfrac{\alpha\, y}{2})$, gdzie α jest zadanym parametrem. Wyjście przyjmuje wartości z przedziału [-1, 1].

Najczęściej stosuje się funkcję aktywacji o ciągłej pochodnej: zamiast funkcji *skoku jednostkowego* (rys. 13.2, lewa strona) stosujemy funkcję *logistyczną* (*sigmoidalną*) (rys. 13.2, prawa strona):

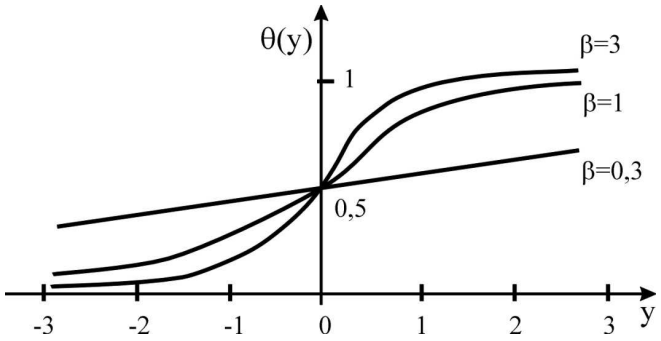


Rys. 13.2:

Dla *logistycznej* (*sigmoidalnej*) funkcji aktywacji o postaci: $z = \theta(y) = \dfrac{1}{1+\exp(-y)}$

pochodna tej funkcji jest postaci: $\boxed{\dfrac{\partial z}{\partial y} = z(1-z)}$

Wpływ wartości parametru β na kształt sigmoidalnej funkcji aktywacji ilustruje rys. 13.3.



Rys. 13.3

13.2. Wielowarstwowy perceptron

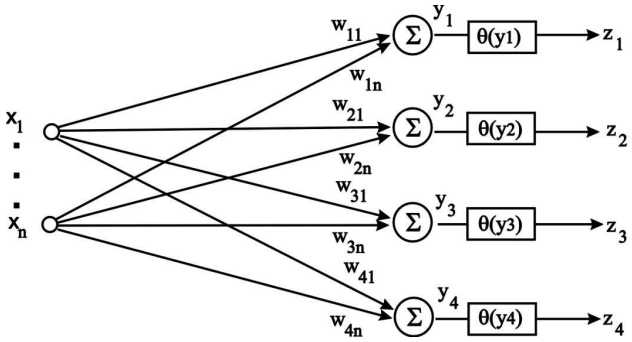
W sieci **jednowarstwowej** neurony ułożone są w jednej warstwie. W sieci typu **perceptron** (sieć **jednokierunkowa**, ang. „*feed-forward*") sygnały wejściowe są przesyłane od wejścia do wyjścia poprzez połączenia **pobudzające**.

13.2.1. Struktura sieci MLP

Połączenie wektora wejściowego z neuronami warstwy wyjściowej jest zwykle pełne co reprezentuje macierz wag połączeń $\mathbf{W}$ (rys. 13.4) Funkcja pobudzenia neuronów warstwy wyjściowej:

$$\mathbf{y} = \mathbf{W} \cdot \mathbf{x}$$

Wielowarstwowy perceptron (ang. *multilayer perceptron*, MLP) posiada warstwę wejściową, warstwę ukrytą (jedną, lub więcej) i neurony warstwy wyjściowej (rys. 13.5).

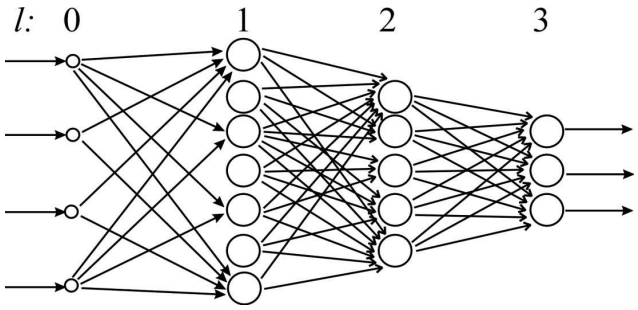


Rys. 13.4

Funkcja aktywacji każdej warstwy o indeksie l (=0,1,2, ...) **wynosi**:

$$\boldsymbol{z}^{(l)} = \theta(\boldsymbol{W}^{(l)}\boldsymbol{z}^{(l-1)} - \boldsymbol{w}_0{}^{(l)})$$

przy czym $\mathbf{z}(0) = \mathbf{x}$, $\theta(\mathbf{y})$ jest najczęściej nieliniową funkcją aktywacji, a $\mathbf{w}_0(l)$ to wektor wag dodatkowych wejść progowych.



Rys. 13.5 Struktura sieci MLP o czterech warstwach

Można pokazać (Cybenko, 1989), że już 3-warstwowy perceptron o sigmoidalnej funkcji aktywacji i n_{hidden} neuronach w warstwie ukrytej, przy $n_{\text{hidden}} \rightarrow \propto$ (dowolnie duża liczba neuronów) może

- aproksymować dowolne zbiory w przestrzeni $\Re^n$, albo
- dowolną funkcję ciągłą zdefiniowaną w tej przestrzeni.

13.2.2. Parametry sieci MLP

Rozważmy sieć o jednym wejściu x, dwóch neuronach ukrytych n_1 i n_2 oraz jednym neuronie wyjściowym n_3 o liniowym wyjściu y_3. Nieliniowa funkcja aktywacji $\theta(y)$ jest postaci tangensa hiperbolicznego. Czyli aproksymujemy funkcję $\mathrm{f}: \mathrm{R} \rightarrow \mathrm{R}$. Opisuje ją wzór:

$$z_3 = y_3 = w_{30} + w_{31} \cdot \theta(w_{10} + w_{11}x) + w_{32} \cdot \theta(w_{20} + w_{21}x)$$

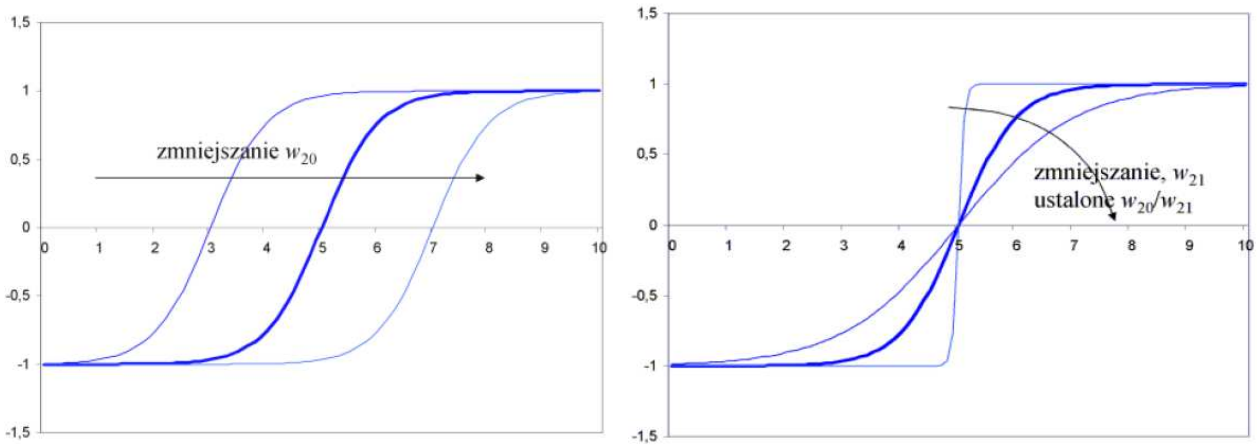
Znaczenie parametrów pojedynczego neuronu warstwy ukrytej jest następujące:

- wagi w_{10}/w_{11}, w_{20}/w_{21} służą do przesuwania wykresu funkcji $\theta(.)$ wzdłuż osi rzędnych.
- wagi w_{11}, w_{21} wpływają na „stromość" wykresu funkcji $\theta(y_1)$, $\theta(y_2)$.

Parametry warstwy ukrytej

Wartości wag w_{10}/w_{11}, w_{20}/w_{21} służą do przesuwania wykresu funkcji $\theta(.)$ wzdłuż osi rzędnych (rys. 13.6, ilustracja lewa).

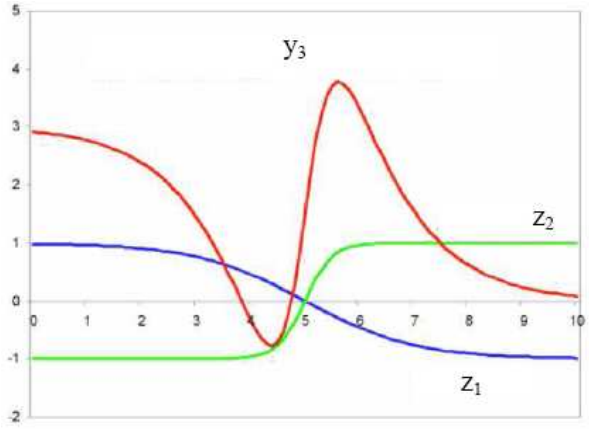
Parametry w_{11}, w_{21} wpływają na „stromość” wykresu funkcji $\theta(y_1)$, $\theta(y_2)$ (rys. 13.6, ilustracja prawa).



Rys. 13.6: Wpływ wartości wag na zachowanie się funkcji

Parametry warstwy wyjściowej

Parametry warstwy wyjściowej służą określeniu stopnia wymieszania wyjść neuronów ukrytych. Im większa wartość wagi w_{31}, wzgl. w_{32} tym większy mnożnik użyty do wykresu wyjścia neuronu 1-szego wzgl. 2-ego (rys. 13.7).



Rys. 13.7

Uczenie z nadzorem. Podstawowa reguła uczenia z nadzorem to **reguła Widrow-Hoffa** (reguła „**delta**”). Według niej waga połączenia *j*-tego wejścia z *i*-tym (neuronem) wyjściowym jest wzmacniana proporcjonalnie do różnicy pożądanej i rzeczywistej aktywacji:

$$\Delta w_{ij} = \mu(s_i - z_i)x_j$$

gdzie s_i to pożądana aktywacja *i*-tego wyjścia.

Jakość aproksymacji. Niech dla wektora wejść $\mathbf{x}_t$ oczekiwany (prawidłowy) wektor wyjściowy perceptronu wynosi $\mathbf{s}_t$. Miarą jakości dla sieci jest suma kwadratów błędów wyjść, $V(\mathbf{W})$ (błędem jest różnica między rzeczywistą a żądaną wartością), uśredniona dla zbioru treningowego:

$$V(\mathbf{W}) = \frac{1}{N_\mathrm{T}} \sum_{t \in \mathrm{T}} \sum_i e_{t,i}^{\ 2} = -\frac{1}{N_\mathrm{T}} \sum_{t \in \mathrm{T}} \sum_i (s_{t,i} - z_{t,i}(y_i(\mathbf{x}_t)))^2$$

13.3. Uczenie sieci MLP

13.3.1. Reguła modyfikacji wag

Uczenie perceptronu polega na optymalizacji wartości *V,* czyli na poszukiwaniu jej minimalnej wartości metodą *spadku w kierunku gradientu* $\nabla V(\mathbf{W})$ (ang. *steepest gradient descent*). Pozwala to w iteracyjny sposób wyznaczyć wartości wag sieci odpowiadające minimum (globalnemu w przypadku sieci liniowej lub najczęściej lokalnemu – dla sieci nieliniowej).

Dla pojedynczej wagi w_{ij} modyfikacja w *t*-tej iteracji jest postaci:

$$w_{ij}(t+1) = w_{ij}(t) - \eta(t)\frac{\partial V(\boldsymbol{W},t)}{\partial w_{ij}}$$

gdzie η oznacza parametr zwany *współczynnikiem uczenia*.

13.3.2. Warstwa wyjściowa

W kolejnej iteracji t wyznaczana jest chwilowa wartość błędu $V(\mathbf{W},t)$ (dla aktualnej próbki uczącej) i wyznaczane są gradienty tej funkcji względem aktualnych wartości wag.

Niech L będzie indeksem warstwy wyjściowej. W tej warstwie gradient aktualnego błędu w iteracji t może być wyrażony bezpośrednio jako funkcja korekty e_t i wejść tej warstwy:

$$\frac{\partial V(\boldsymbol{W},t)}{\partial w_{ij}^{(L)}} = e_{t,i}\cdot\frac{\partial e_{t,i}}{\partial w_{ij}^{(L)}} = 2e_{t,i}\cdot z'_i(y_i)\cdot x_j^{(L)} = 2(z_i - s_i)\cdot z'_i(y_i)\cdot x_j^{(L)}$$

gdzie $$z'_i(y_i) = \frac{dz_i}{dy_i}$$

oznacza pochodną funkcji aktywacji neuronu, a $x_j^{(L)}$ oznacza j-te wejście dla tej warstwy (czyli wyjście warstwy poprzedniej $z_j^{(L-1)}$).

Stąd wynika reguła modyfikacji wagi w warstwie wyjściowej L sieci :

$$w_{ij}^{(L)}(t+1) = w_{ij}^{(L)}(t) - \eta\cdot(z_i - s_i)\cdot z'_i(y_i)\cdot x_j^{(L)}$$

Oznaczmy korektę wartości pobudzenia neuronu wyjściowego przez

$$\delta_i^{(L)} = e_i\cdot z'_i(y_i)$$

tzn. aktualny błąd dla *i*-tego wyjścia przeskalowany zostaje przez pochodną funkcji aktywacji (indeks iteracji pomijamy). Reguła modyfikacji wagi przyjmuje wtedy postać:

$$w_{ij}^{(L)}(t+1) = w_{ij}^{(L)}(t) - \eta\cdot\delta_i^{(L)}\cdot x_j^{(L)}$$

13.3.3. Warstwy ukryte

J-te wyjście z poprzedniej warstwy, $x_j^{(L)} = z_j^{(L-1)}$, jest połączone wektorem wag $(\mathbf{w}_j^{(L)})^T$ z warstwą wyjściową. Możemy teraz zrzutować sumaryczną korektę wymaganą dla neuronów warstwy wyjściowej na poprzednią warstwę. Otrzymujemy korektę pobudzenia neuronu warstwy ukrytej:

$$\delta_j^{(L-1)} = \sum_i w_{ij}^{(L)}\delta_i^{(L)}\cdot z_j^{(L-1)\prime}(y_j)$$

Wystarczy zauważyć, że dla obliczenia gradientu funkcji aktualnego błędu $V(\mathbf{W},t)$ względem wag warstwy ukrytej sieci wystarczy informacja o korekcie pobudzenia neuronów warstwy, która po niej następuje. Reguła modyfikacji wag w warstwie ukrytej o indeksie $l = 1,...,L\text{-}1,$ jest postaci:

$$w_{ij}^{(l)}(t+1) = w_{ij}^{(l)}(t) - \eta \cdot \delta_i^{(l)} \cdot x_j^{(l)}$$

Z tej przyczyny podstawowa zasada modyfikacji wag sieci MLP w procesie uczenia sieci jest nazywana **regułą wstecznej propagacji błędu** (ang. *error backpropagation rule*) – błąd propaguje się w kierunku od wyjścia do wejścia sieci.

Ogólna postać **reguły wstecznej propagacji błędu** dla macierzy wag warstwy l wynosi:

$$\Delta \boldsymbol{W}^{(l)} = \eta \cdot \Delta^{(l)} \cdot \left[x^{(l)}\right]^T, \quad x^{(1)} = x$$

dla $l = 1, \ldots, L$; gdzie $\boldsymbol{\Delta}^{(l)}$ jest wektorem korekcji pobudzeń neuronów l-tej warstwy.

Modyfikacja wag rozpoczyna się od ostatniej warstwy $(l = L)$ i przemieszcza się wstecz warstwa po warstwie, aż zakończy się na warstwie o indeksie l=1.

Podczas kroku modyfikacji wag propagowane są „wstecz" wartości korekty, obliczone początkowo dla najwyższej warstwy $(l = L,\ldots,1)$:

$$\Delta^{(L)} = (\boldsymbol{z}^{(L)} - \boldsymbol{s}).* \left[\frac{\partial \theta}{\partial y}\right]$$

a następnie dla kolejnych warstw ukrytych:

$$\Delta^{(l-1)} = (\boldsymbol{W}^{(l)})^T \Delta^{(l)}.* \left[\frac{\partial \theta}{\partial y}\right]$$

Uwaga: symbol **.*** we wzorach na korektę wag oznacza mnożenie elementu pierwszego wektora z odpowiednim elementem drugiego wektora (element-po-elemencie).

W szczególności przy logistycznej funkcji aktywacji wartości korekty, dla $l = L,...,1$, obliczamy jako:

$$\Delta^{(L)} = (\mathbf{z}^{(L)} - \mathbf{s}).* \left[1 - \mathbf{z}^{(L)}\right] * \mathbf{z}^{(L)}$$

$$\Delta^{(l-1)} = (\mathbf{W}^{(l)})^T \Delta^{(l)}.* \left[1 - \mathbf{z}^{(l-1)}\right] * \mathbf{z}^{(l-1)}$$

13.4. Pytania

1. Omówić model neuronu.
2. Omówić pojęcie wielowarstwowego perceptronu MLP.
3. Jak wpływa liczba warstw i wagi sieci na jakość aproksymacji?
4. Omówić podstawowy algorytm uczenia sieci MLP.

13.5. Zadania

Zad. 13.1

Pojedynczy neuron może realizować liniowe funkcje logiczne. Przedstawić budowę pojedynczego neuronu implementującego funkcje logiczne (Boole'a): AND, OR, NOT.

Przedstawić budowę sieci implementującej funkcję XOR.

Zad. 13.2

Wykonać jedną pełną iterację procesu uczenia 3-warstwowego perceptronu (jedna warstwa ukryta), z 3 neuronami w warstwie ukrytej i po dwóch wejściach i wyjściach. Uczenie przebiega według podstawowej metody wstecznej propagacji błędu, przy następujących założeniach:

- stały współczynnik uczący = 0.1,
- *sigmoidalna* funkcja aktywacji, z = $\theta(y) = \dfrac{1}{1+\exp(-y)}$
- początkowe wartości wag w macierzach $\mathbf{W}^{(1)}$ i $\mathbf{W}^{(2)}$ wynoszą

$$\boldsymbol{W}^{(1)} = \begin{bmatrix} 1 & -1 \\ 1 & 1 \\ -1 & 1 \end{bmatrix} \qquad \boldsymbol{W}^{(2)} = \begin{bmatrix} 1 & -1 & 1 \\ -1 & 1 & -1 \end{bmatrix}$$

- pierwsza próbka ucząca to: $\mathbf{x}$=(1,2); $f(\mathbf{x})$ = (1, 0).

Dla ułatwienia można przybliżyć wartości funkcji aktywacji zgodnie z wartościami w poniższej tabelce:

y	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0	1.1
z=θ(y)	0.525	0.55	0.575	0.60	0.623	0.646	0.668	0.69	0.71	0.73	0.75

y	-1	-0.9	-0.8	-0.7	-0.6	-0.5	-0.4	-0.3	-0.2	-0.1	0.0
z=θ(y)	0.27	0.29	0.31	0.33	0.35	0.375	0.40	0.425	0.45	0.475	0.50

Bibliografia

[1] J. Arabas, P. Cichosz, A, Dydyński: *Inteligentne techniki obliczeniowe*. OKNO, Politechnika Warszawska, 2005. e-skrypt.

[2] S. Russell, P. Norvig: *Artificial Intelligence: A Modern Approach*. New Jersey, Prentice Hall, 2002, 2010

[3] M. Flasiński M.: *Wstęp do Sztucznej Inteligencji*. Wydawnictwo Naukowe PWN, Warszawa, 2011.

[4] R. Rutkowski R.: *Metody i techniki sztucznej inteligencji*. Warszawa, Wydawnictwo Naukowe PWN, 2005.

[5] W. Traczyk W.: *Inżynieria Wiedzy*. Oficyna Wydawnicza EXIT, Warszawa, 2010.