Przemysław Klęsk
Joanna Kołodziejczyk
Marcin Pietrzykowski
Jacek Klimaszewski

Sztuczna inteligencja

Zachodniopomorski
Uniwersytet Technologiczny
w Szczecinie

Przemysław Klęsk
Joanna Kołodziejczyk
Marcin Pietrzykowski
Jacek Klimaszewski

Sztuczna inteligencja

Szczecin 2023

Autorzy:
Przemysław Klęsk, Joanna Kołodziejczyk, Marcin Pietrzykowski, Jacek Klimaszewski.

Recenzent:
...

Opracowanie redakcyjne:
...

Skład komputerowy:
Przemysław Klęsk, Joanna Kołodziejczyk, Marcin Pietrzykowski, Jacek Klimaszewski.

Wydano za zgodą Rektora Zachodniopomorskiego Uniwersytetu Technologicznego w Szczecinie

Skrypt został przygotowany w ramach Projektu: „ZUT 2.0 – Nowoczesny Zintegrowany Uniwersytet" współfinansowanego przez Unię Europejską w ramach Europejskiego Funduszu Społecznego;

ISBN 978-83-7663-???-?

Wydawnictwo Uczelniane Zachodniopomorskiego Uniwersytetu Technologicznego w Szczecinie
70-311 Szczecin, al. Piastów 48
tel. 91 449 47 60
e-mail: wydawnictwo@zut.edu.pl

Spis treści

Przedmowa

Przeznaczenie i cele skryptu

Niniejszy skrypt jest przeznaczony dla studentów kierunku informatyka lub kierunków pokrewnych. Został opracowany jako wsparcie dydaktyczne dla przedmiotu *„Sztuczna inteligencja”*, który jest prowadzony na Wydziale Informatyki Zachodniopomorskiego Uniwersytetu Technologicznego w Szczecinie na rzecz studentów pierwszego stopnia (inżynierskiego). Przedmiot ten ma charakter elementarny — stanowi wprowadzenie w podstawowe zagadnienia i algorytmy z zakresu sztucznej inteligencji, i nie zakłada żadnej wiedzy ze strony studenta w tym zakresie. Zakłada się natomiast posiadanie podstawowej wiedzy i umiejętności z: algorytmiki, struktur danych, programowania (w tym programowania obiektowego), matematyki na poziomie akademickim uczelni technicznych. W kolejnych częściach skryptu prezentowane są zagadnienia dotyczące: przeszukiwania grafów i drzew gier dwuosobowych, optymalizacji stochastycznej, podstaw uczenia maszynowego, oraz systemów wnioskujących i reprezentacji wiedzy.

W zamierzeniu autorów skrypt ma stanowić wsparcie zarówno dla wykładowej jak i laboratoryjnej formy kursu. Treści zawarte w skrypcie zostały przygotowane na podstawie wiedzy i doświadczeń pracowników Katedry Sztucznej Inteligencji i Matematyki Stosowanej prowadzących zajęcia ze sztucznej inteligencji na prze-

strzeni około 15 lat, a w szczególności na podstawie materiałów dydaktycznych gromadzonych na stronie `http://wikizmsi.zut.edu.pl`.

W poszczególnych rozdziałach po treściach teoretycznych następują zestawy ćwiczeń i instrukcji opracowane z myślą o zajęciach laboratoryjnych. W przypadku większości z tych instrukcji zamierzono użycie konkretnego języka programowania, bibliotek czy też pewnego API opracowanego przez osoby prowadzące zajęcia. Wówczas dany zestaw ćwiczeń został opatrzony odpowiednim nagłówkiem, w szczególności:

- **Java + biblioteka *SaC*** (autorzy: P. Klęsk, M. Korzeń),
- **C# + biblioteka *AISearch*** (autor: M. Pietrzykowski),
- **C++ + biblioteka *SI++*** (autor: J. Klimaszewski),
- **MATLAB**,
- **Python**.

Niemniej, na problemy sformułowane w tych instrukcjach można w większości przypadków także patrzeć na ogólnym poziomie algorytmicznym, bez związku z konkretnym językiem programowania. Innymi słowy pozostawiamy osobom prowadzącym zajęcia praktyczne pewną swobodę przy wyborze środowiska pracy, przy czym sugerowane są raczej środowiska wysokopoziomowe z możliwościami szybkiego prototypowania, pracy na macierzach i kreślenia wykresów, takie jak np.: Python, MATLAB, Wolfram Mathematica, R.

Wykorzystane oprogramowanie

Niniejszy skrypt został od strony edycyjnej złożony w systemie LaTeXz wykorzystaniem szablonu stylu przeznaczonego dla Wydziału Informatyki, ZUT w Szczecinie, który został opracowany przez Joannę Kołodziejczyk.

SaC — biblioteka do przeszukiwania napisana w języku Java

SaC (ang. *Search and Conquer*) jest obiektową biblioteką napisaną przez Przemysława Klęska i Marcina Korzenia w języku Java na potrzeby przeszukiwań grafów oraz drzew gier. Pierwsza wersja biblioteki powstała w latach 2012–2013 w ramach większego projektu akademickiego o nazwie TEWI[1] finansowanego z Europejskiego Funduszu Rozwoju Regionalnego (POIG.02.03.00-00-028/09). Bibliotekę można pobrać z repozytorium na GitHubie: `https://github.com/pklesk/sac`, a oficjalną stronę biblioteki znaleźć pod adresem: `https://pklesk.github.io/sac`. W szczególności na stronie umieszczono podręcznik użytkownika (z przykładami kodów źródłowych i licznymi ilustracjami[2]): `https://github.com/`

[1] Telekomunikacja Edukacja Wiedza Innowacje

[2] Ilustracje generowane z pomocą oprogramowania *Graphviz*, `http://www.graphviz.org`.

pklesk/sac/releases/download/1.0.3/sac-1.0.3-userguide.pdf. Biblioteka obejmuje łącznie 9 gotowych algorytmów przeszukujących wraz z odpowiednimi strukturami danych i ustawieniami konfiguracyjnymi.

AISearch — biblioteka do przeszukiwania napisana w języku C#

AISearch jest małą biblioteką autorstwa Marcina Pietrzykowskiego napisaną w języku C# zawierającą implementacje podstawowych algorytmów używanych w przeszukiwaniu grafów oraz przeszukiwaniu drzew gier. Bibliotekę można pobrać z repozytorium na GitHubie: `https://github.com/mpietrzykowski/AISearch`. Pobrane rozwiązanie programu Visual Studio (*Solution*) zawiera dwa projekty (*Project*):

- `AISearch` — projekt zawierający właściwą bibliotekę,
- `Exercise` — przykładowy projekt aplikacji konsolowej korzystającej z *AISearch*.

Więcej informacji na temat tej biblioteki zamieszczono w dodatku 11.

SI++ — biblioteka do przeszukiwania napisana w języku C++

SI++ jest małą biblioteką autorstwa Jacka Klimaszewskiego napisaną w języku C++, która zawiera struktury danych i implementacje algorytmów koniecznych do wykonania ćwiczeń. Aby z niej korzystać, trzeba użyć kompilatora zgodnego ze standardem C++17. Bibliotekę można pobrać z repozytorium w serwisie GitHub: `https://github.com/Szachista/SIplusplus`.

I Przeszukiwanie

1. O przeszukiwaniu ogólnie...

Algorytmy przeszukujące stanowią znaczącą część badań w ramach sztucznej inteligencji. Patrząc z perspektywy historycznej na rozwój tej dziedziny, *znajdowanie ścieżek* i *szachy* przychodzą na myśl jako prawdopodobnie dwa najbardziej naturalne przykłady zadań, gdzie algorytmy przeszukujące stosowano wielokrotnie i z dużym z powodzeniem. Te przykłady są także dobrymi reprezentatami dwóch ogólnych grup problemów związanych z przeszukiwaniem, które są omawiane w tym rozdziale:

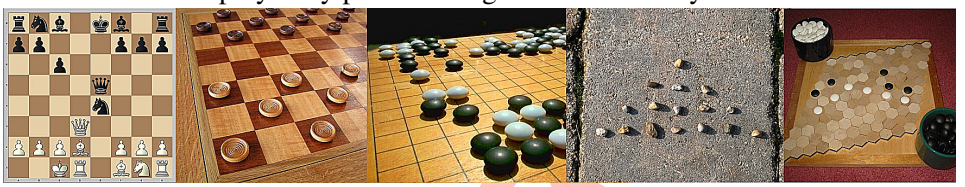
1. **problemy optymalizacji dyskretnej (kombinatorycznej)**, gdzie dany problem można przedstawić jako *graf stanów*, a znalezienie rozwiązania sprowadza się do przeszukiwania;
2. **problemy gier dwuoosobowych**, gdzie poszukuje się najlepszego ruchu lub decyzji w pewnej grze lub sytuacji konflikowej, którą można przedstawić jako *drzewo gry*.

Rys. 1.1 przedstawia ilustracyjnie dwie powyższe grupy.

Pierwsza grupa zawiera m.in. grafy, które można interpretować geograficznie lub fizycznie, gdzie pewien obiekt przemieszcza się pomiędzy dostępnymi lokalizacjami. Można tu myśleć np. o: znajdowaniu najkrótszych ścieżek dla różnych środków transportu, problemach routingu, pokonywaniu labiryntów itp. Z drugiej strony grupa ta obejmuje również grafy (o których należy myśleć bardziej abs-

przykłady problemów optymalizacji dyskretnej (kombinatorycznej):

przykłady problemów gier dwuoosobowych:



Rys. 1.1: Ilustracja dwóch ogólnych grup problemów, które są rozwiązywane przy wykorzystaniu algorytmów przeszukujących (źródło: *Google Images*).

trakcyjnie) związane z rozmaitymi układankami lub łamigłówkami, gdzie poprzez pewne *manipulacje* możemy zmieniać stan obiektu. Zwykle chcielibyśmy odkryć taką sekwencję tych manipulacji, która przeprowadza obiekt w stan o pewnych pożądanych własnościach, który rozumiemy jako rozwiązanie. Można tu wymienić przykłady czysto rekreacyjne (puzzle przesuwne, kostka Rubika, sudoku, pasjansy), ale także przykłady bardziej praktyczne i techniczne: problemy upakowań przestrzennych (dwu- i trójwymiarowe), optymalizację cięcia materiałów, układanie harmonogramów, planowanie zasobów itp.

Jeśli chodzi o drugą grupę, to oczywistymi przykładami dla niej są gry umysłowe: szachy, warcaby, GO, Hex, Nim, Rój, Connect4, kółko i krzyżyk, i wiele innych. Poza nimi do grupy tej zaliczyć można przykłady bliższe teorii gier, jak np. słynny dylemat więźnia, problemy negocjacyjne, konflikty polityczne, wojny konkurencji. W ustalonej pozycji w grze (stanie gry) gracz ma zwykle do dyspozycji pewną liczbę *ruchów*, które przenoszą grę w nowe pozycje. W każdej z nich przeciwnik ma do dyspozycji pewną liczbę kontrruchów i taki schemat jest kontynuowany, generując w sposób naturalny strukturę *drzewa*. Niestety, drzewa gier rozrastają się w tempie wykładniczym, stąd też programy komputerowe w praktyce muszą ograniczyć się do przejrzenia tylko fragmentu takiego drzewa przed podjęciem pojedynczej decyzji. Dla typowych gier analiza komputerowa obejmuje zwykle kilka lub kilkanaście poziomów drzewa. Przypisując pewne liczbowe oceny pozycjom końcowym (liściom w drzewie), które są odległymi konsekwencjami ruchów na poziomie korzenia, algorytm jest w stanie propagować te oceny odpowiednio w górę drzewa, a następnie wskazać najbardziej obiecujący ruch.

Warto także wspomnieć, że nie tylko gry umysłowe mogą być analizowane w powyższy sposób. Wiele gier komputerowych (włączając w to także gry zręcznościowe lub nawet „strzelanki") pozwala często osadzić w sobie elementy sztucznej inteligencji oparte na drzewie gry. Istotnym warunkiem jest to, abyśmy umieli wyróżnić dla danego agenta (postaci, bota) pewien skończony zbiór możliwych akcji (ruchów) w chwilach podejmowania przez niego decyzji.

Z perspektywy programisty różne algorytmy przeszukujące mają bardzo wiele wspólnych elementów. Stąd też naturalnym pomysłem jest próba wyabstrahowania pewnego zbioru interfejsów i klas, czyli pewnego API, które ujednolicałoby realizację tych algorytmów. Przykładami takiego API na rzecz niniejszego skryptu są biblioteki: *SaC*, *AISearch*, *SI++*. Chodzi tu także o to, aby użytkownik (programista) mógł w sposób prosty zdefiniować swój problem w terminach danego API i uruchomić przeszukiwanie z możliwością łatwego przełączania się pomiędzy różnymi: algorytmami, wariantami funkcji oceny, nastawami struktur danych itp.

Centralną rolę w algorytmach przeszukujących pełni byt określany mianem ***stanu*** (ang. *state* lub *search state*). O stanach możemy mówić zarówno w kontekście przeszukiwania grafów (wówczas stany utożsamiane są z wierzchołkami grafu, a krawędzie z manipulacjami), jak i w kontekście drzew gier (wówczas stany utożsamiane są z pozycjami w grze, a krawędzie z ruchami). Stan może reprezentować np.: częściowo wypełnioną planszę sudoku, układ kart na stole w trakcie układania pasjansa, pomieszaną kostkę Rubika na pewnym etapie rozwiązywania, pewną pozycję szachową, przesiadkę podróżnika na konkretnej stacji i o konkretnej godzinie, częściowe upakowanie towarów w magazynie itp.

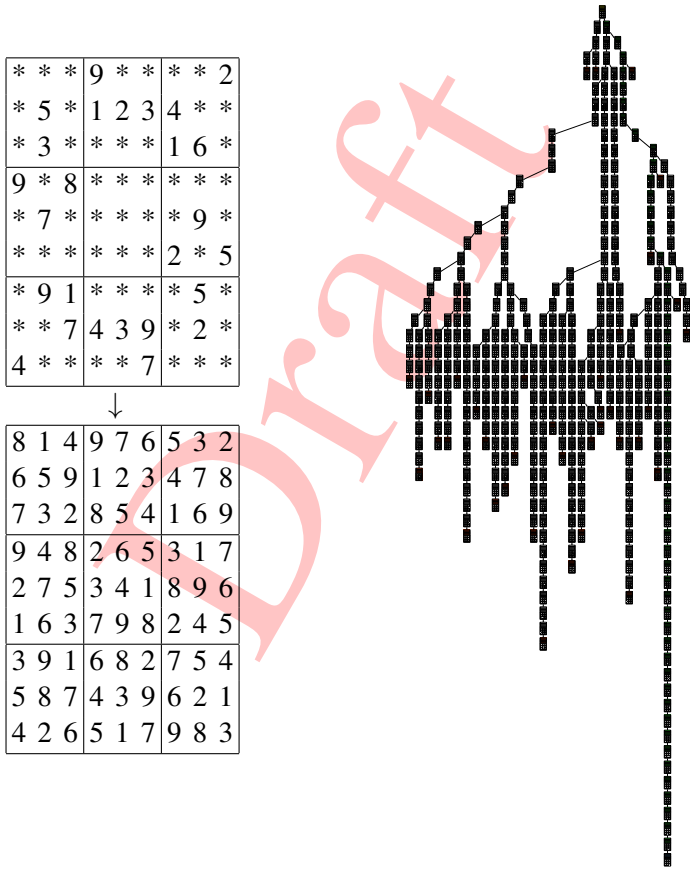
Dokładne informacje opisujące statyczną zawartość stanu musi oczywiście zapewnić programista. Są to informacje specyficzne dla danego problemu. Sama zaś „mechanika" przeszukiwania jest niezależna od problemu i można w niej wyróżnić pewne kluczowe powtarzające się elementy:

1. **Generowanie stanów potomych** — *Jakie nowe stany (bezpośredni potomkowie) mogą zostać wygenerowane z danego stanu?*
2. **Identyfikacja** — *Jakie identyfikatory (napisowe lub całkowitoliczbowe) mogą zostać przypisane do stanów, tak aby ten sam stan nie był odwiedzany niepotrzebnie wielokrotnie?*
3. **Zakończenie (warunek stopu)** — *Czy dany stan jest stanem końcowym, tj. rozwiązaniem (grafy) lub stanem zwycięskim (drzewa gier)?*

3'. **Funkcja oceny / heurystyka**[1] **(opcjonalnie)** — *Oszacowanie, jak daleko dany stan jest od rozwiązania (grafy) lub ocena, w jakim stopniu dany stan reprezentuje przewagę gracza maksymalizującego lub minimalizującego (drzewa gier).*

[1] heurystyka (gr. *heuresis* — odnaleźć, odkryć, *heureka* — znalazłem)

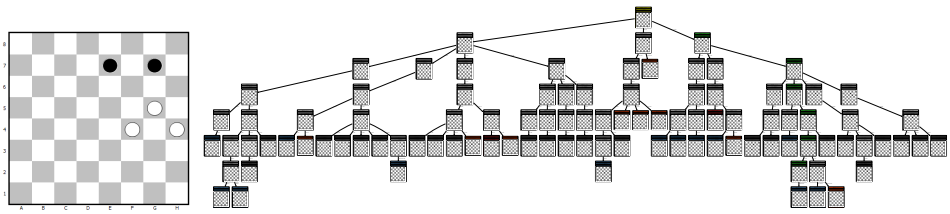
Ostatni element oznaczono numerem 3', ponieważ może on być tak naprawdę postrzegany jako rozszerzenie elementu 3. Mowa w nim o pewnej funkcji oceniającej, nazywanej często funkcją heurystyczną lub krótko heurystyką. W zależności od rodzaju rozpatrywanego zadania przeszukiwania (grafy czy gry) dokładny sens terminu heurystyka będzie nieco inny. Niemniej, w każdym z tych przypadków heurystyka dostarczy pewnej przybliżonej i racjonalnej informacji, która ukierunkuje przeszukiwania na stany istotne, pozwalając zmniejszyć wysiłki obliczeniowe marnowane na stany jałowe.



Rys. 1.2: Przykładowa łamigłówka sudoku oraz graf przeszukiwań wygenerowany przez algorytm *Best-first search* używający funkcji heurystycznej „suma pozostałych możliwości" (źródło: *opracowanie własne*).

Rysunki 1.2 i 1.3 ilustrują przykładowy graf i przykładowe drzewo przeszukiwań wygenerowane przez algorytmy omawiane w tym rozdziale odpowiednio dla łamigłówki sudoku oraz końcówki warcabowej. Zachęcamy czytelnika do

wykonania powiększeń tych ilustracji i obejrzenia szczegółów.



Rys. 1.3: Przykładowa końcówka warcabowa (rozpoczynają białe) wraz z przykładowym drzewem gry algorytmu *przycinanie α-β* (źródło: *opracowanie własne*).

Uwaga: w niniejszym skrypcie używany jest powszechnie termin *stan* (w związku z rozważanymi problemami i algorytmami przeszukiwania pewnych przestrzeni stanów) jako równoważnik terminów *wierzchołek* lub *węzeł*, które są stosowane dla grafów i drzew w bardziej ogólnym i klasycznym nazewnictwie algorytmicznym.

2. Przeszukiwanie grafów

Większość algorytmów do przeszukiwań grafowych można wygodnie sformułować z użyciem dwóch zbiorów (kolekcji) stanów — nazywanych zwyczajowo zbiorami: *Open* i *Closed*. W danym momencie pracy algorytmu zbiór *Closed* zawiera stany, które zostały już odwiedzone (i okazały się nie być rozwiązaniem), podczas gdy zbiór *Open* zawiera stany oczekujące do odwiedzenia. Stany oczekujące zostają wygenerowane jako potomkowie (lub inaczej — grafowi sąsiedzi) stanów poprzednio odwiedzonych.

W zależności od rodzaju algorytmu przeszukującego, zbiory *Open* i *Closed* są implementowane z wykorzystaniem różnych struktur danych, co przekłada się na ich zachowanie i wydajność. Rzeczą decydującą o tym, z jakim algorytmem mamy tak naprawdę do czynienia, jest porządek, według którego wygenerowane stany są pobierane (i usuwane) ze zbioru *Open*.

W tym rozdziale przedstawiamy sześć wybranych algorytmów przeszukiwania grafów, ze szczególnym naciskiem na algorytmy zaliczane do grupy przeszukiwań *poinformowanych*. Znajdują one najczęstsze zastosowanie w ramach zadań związanych ze sztuczną inteligencją, a kluczowym elementem do ich skutecznego działania jest opracowanie odpowiedniej funkcji kosztu, nazywanej zwyczajowo funkcją *heurystyczną*.

2.1 Przeszukiwanie niepoinformowane (ślepe)

2.1.1 Breadth-first i depth-first search

W przypadku tych dwóch metod ciężko jest o wskazanie ich faktycznych autorów. Właściwie traktuje się je bardziej jako proste techniki *przechodzenia* grafu (przechodzenie wyczerpujące i ślepe), aniżeli faktyczne algorytmy przeszukujące. Od takich wymaga się raczej, aby były poinformowane, tzn. wiedzione pewną użyteczną informacją decydującą o kolejności odwiedzania stanów. Breadth-first i depth-first search należy rozumieć odpowiednio jako przechodzenie grafu *wszerz* i *w głąb*. Jest prawdopodobnym, że historycznie pierwsza wersja przechodzenia grafu w głąb była badana już w XIX w. przez francuskiego matematyka Charlesa Pierre'a Trémaux jako strategia rozwiązywania labiryntów [Shi11, s. 46–48].

Niech s_0 oznacza stan początkowy. Jak wskazują nazwy, w podejściu breadth-first algorytm musi odwiedzić wszystkie stany na głębokości d (tzn. oddalone o d krawędzi od s_0), zanim może przystąpić do odwiedzania stanów na głębokości $d+1$. I tak dla każdego d. W pewnym sensie przeciwnie, w podejściu depth-first algorytmowi nie wolno odwiedzić nieodwiedzonych jeszcze stanów na głębokości d, jeżeli istnieją pewne wygenerowane i nieodwiedzone jeszcze stany na głębokości $d+1$. Jeżeli krawędzie grafu posiadają pewne wagi (koszty przejść), to są one ignorowane w obu omawianych podejściach. Istotna jest tylko głębokość, czyli liczba przejść pomiędzy stanem początkowym s_0, a danym stanem.

Poniższe pseudokody prezentują algorytmy breadth-first i depth-first search. Jak można zauważyć, większość kroków algorytmicznych pozostaje taka sama. Jedyną różnicą jest porządek stanów pobieranych ze zbioru *Open* (linia nr 6).

Algorytm 1 Breadth-first search

```
1:  procedura BreadthFirstSearch(s_0)                    ▷ stan początkowy: s_0
2:      Closed := ∅                                      ▷ pusty zbiór odwiedzonych stanów
3:      ustaw pusty wskaźnik rodzica dla s_0
4:      Open := {s_0}                                    ▷ kolejka stanów oczekujących
5:      dopóki Open ≠ ∅ wykonaj
6:          pobierz (i usuń) z Open stan s o najmniejszej głębokości
7:          jeżeli s jest stanem końcowym to zwróć s     ▷ znaleziono rozwiązanie
8:          wygeneruj zbiór stanów {t} potomnych dla s   ▷ ustawiając wskaźniki na rodzica s
9:          dla wszystkich t wykonaj
10:             jeżeli t ∉ Closed i t ∉ Open to dodaj t do Open
11:         dodaj s do Closed
12:     zwróć wynik pusty                               ▷ nie znaleziono rozwiązania
```

W powyższych zapisach zakładamy, że stany są wyposażone w informację o swoim rodzicu oraz swojej głębokości, tzn. na poziomie implementacji każdy obiekt reprezentujący stan jest wyposażony we wskaźnik (lub referencję) na stan

Algorytm 2 Depth-first search

```
1:  procedura DepthFirstSearch(s_0)                          ▷ stan początkowy: s_0
2:      Closed := ∅                                          ▷ pusty zbiór odwiedzonych stanów
3:      ustaw pusty wskaźnik rodzica dla s_0
4:      Open := {s_0}                                        ▷ kolejka stanów oczekujących
5:      dopóki Open ≠ ∅ wykonaj
6:          pobierz (i usuń) z Open stan s o największej głębokości
7:          jeżeli s jest stanem końcowym to zwróć s         ▷ znaleziono rozwiązanie
8:          wygeneruj zbiór stanów {t} potomnych dla s       ▷ ustawiając wskaźniki na rodzica s
9:          dla wszystkich t wykonaj
10:             jeżeli t ∉ Closed i t ∉ Open to dodaj t do Open
11:         dodaj s do Closed
12:     zwróć wynik pusty                                   ▷ nie znaleziono rozwiązania
```

rodzicielski oraz pole całkowite przechowujące głębokość. Stan początkowy s_0 ma pusty wskaźnik na rodzica (`null`). Gdy pewien stan potomny t zostaje wygenerowany na podstawie stanu s, to głębokość stanu t jest ustalana na głębokość s plus jeden.

Mając na uwadze porządek odwiedzania stanów, do implementacji zbioru *Open* można użyć odpowiednio standardowej kolejki FIFO (First In First Out) do przechodzenia wszerz oraz struktury LIFO (Last In First Out) — czyli inaczej stosu — do przechodzenia w głąb.

2.1.2 Algorytm Dijkstry

W 1956 r. Edsgar Dijkstra zaproponował w pracy [Dij59] algorytm do znajdowania najkrótszych ścieżek w grafie z wagami. W oryginalnym sformułowaniu algorytm ten znajdował *wszystkie* najkrótsze ścieżki pomiędzy ustalonym stanem początkowym a *wszystkimi* pozostałymi stanami (ang. *single source all shortest paths*). To wyjaśnia, dlaczego algorytm Dijkstry poza wagami *nie* używa żadnej dodatkowej informacji (np. informacji heurystycznej szacującej pozostały koszt do celu), ponieważ nie można wskazać pojedynczego celu. Tym samym algorytm Dijkstry jest uznawany także za algorytm przeszukiwania niepoinformowanego, podobnie jak breadth-first i depth-first search.

Algorytm Dijkstry może zostać w prosty sposób zmodyfikowany, tak aby zatrzymywał się wcześniej (zanim ustalone zostaną wszystkie najkrótsze ścieżki), w momencie gdy napotka na pewien stan wyróżniony jako końcowy. Poniżej użyjemy tego właśnie wariantu w celu pewnego ujednolicenia zapisów algorytmicznych dla wszystkich technik przeszukujących prezentowanych w niniejszym skrypcie. Dodatkowo, jak się później okaże, algorytm Dijkstry sformułowany w wariancie z jednym stanem docelowym będzie mógł być postrzegany jako szczególny

przypadek algorytmu A* opisanego dalej[1].

Niech $g(s)$ oznacza dokładny koszt związany z przebyciem drogi od stanu początkowego s_0 do stanu s. W zależności od aplikacji koszt może reprezentować np.: przebytą odległość, czas podróży, zużytą energię itp. Dalej, niech $\Delta(s \to t)$ oznacza koszt przejścia ze stanu s do stanu t, gdzie t jest bezpośrednim potomkiem (sąsiadem) s. Algorytm Dijkstry możemy sformułować w sposób następujący z wykorzystaniem tych wielkości.

Algorytm 3 Algorytm Dijkstry

```
1:  procedura DIJKSTRA(s_0)                                   ▷ stan początkowy: s_0
2:      Closed := ∅                                           ▷ pusty zbiór odwiedzonych stanów
3:      g(s_0) := 0                                           ▷ koszt przebyty od startu
4:      ustaw pusty wskaźnik rodzica dla s_0
5:      Open := {s_0}                                         ▷ kolejka stanów oczekujących
6:      dopóki Open ≠ ∅ wykonaj
7:          pobierz (i usuń) z Open stan s o najmniejszej wartości g(s)   ▷ operacja „poll"
8:          jeżeli s jest stanem końcowym to zwróć s          ▷ znaleziono rozwiązanie
9:          wygeneruj zbiór stanów {t} potomnych dla s
10:         dla wszystkich t wykonaj
11:             jeżeli t ∈ Closed to kontynuuj od kolejnej iteracji   ▷ t już odwiedzone
12:             g(t) := g(s) + Δ(s → t)
13:             ustaw wskaźnik rodzica t na s
14:             jeżeli t ∉ Open to
15:                 dodaj t do Open
16:             w przeciwnym razie
17:                 jeżeli nowy koszt g(t) jest mniejszy niż znany dotychczas to
18:                     zastąp t w Open nowym egzemplarzem (aktualnie badanym)
19:                     uaktualnij pozycję t w Open
20:         dodaj s do Closed
21:     zwróć wynik pusty                                    ▷ nie znaleziono rozwiązania
```

Poniższe twierdzenie zapewnia, że algorytm Dijkstry w wariancie z jednym stanem docelowym (Algorytm 3) znajduje ścieżkę najkrótszą do tego stanu.

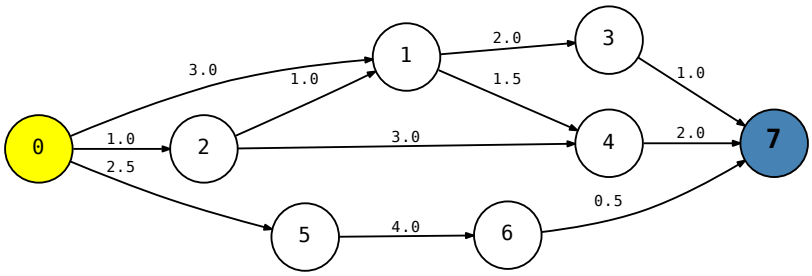
Twierdzenie 2.1.1 Niech s^* oznacza stan docelowy. Jeżeli istnieje przynajmniej jedna ścieżka pomiędzy s_0 a s^*, to algorytm Dijkstry (Algorytm 3) gwarantuje znalezienie ścieżki najkrótszej — tj. ścieżki o najmniejszej wartości kosztu g.

Dowód. Ponieważ istnieje przynajmniej jedna ścieżka prowadząca do s^*, to Algorytm 3 zatrzyma się (linia 8) zwracając pewien egzemplarz s^* o koszcie $g(s^*)$. O wszystkich stanach s przebywających w zbiorze *Open* w chwili stopu wiadomo,

[1] Wymuszenie wartości zero dla składnika heurystycznego w funkcji oceny algorytmu A* powoduje, że algorytm A* redukuje się do algorytmu Dijkstry.

że $g(s) \geqslant g(s^*)$. Jednocześnie wiadomo, że wszystkie stany osiągalne z s_0 ścieżkami o koszcie mniejszym niż $g(s^*)$ musiały już zostać zbadane (ze względu na pobieranie stanu o najmniejszym koszcie w każdym kroku pętli) i żadna z nich nie prowadziła do rozwiązania (nie zaszedł warunek stopu w linii nr 8). ■

Dla zilustrowania sposobu działania przeszukiwań ślepych na rys. 2.1 przedstawiono prosty graf z wagami. Stan początkowy jest oznaczony kolorem żółtym, stan końcowy niebieskim. Algorytmy breadth-first i depth-first search uruchomione



Rys. 2.1: Przykładowy graf z wagami. Stan początkowy oznaczony kolorem żółtym, stan końcowy niebieskim.

na rzecz tego grafu będą oczywiście ignorowały wagi (lub równoważnie traktowały koszt każdej krawędzi jako równy 1). Algorytm Dijkstry będzie kierował się minimalną sumą wag i wykryje najkrótszą ścieżkę $(0,2,1,3,7)$ o łącznym koszcie 5.0. Porządek odwiedzanych stanów dla poszczególnych algorytmów będzie następujący:

$$\begin{aligned} \text{breadth-first search (przejście wszerz):} &\quad (0,1,2,5,3,4,6,7), \\ \text{depth-first search (przejście w głąb):} &\quad (0,1,3,7), \\ \text{algorytm Dijkstry:} &\quad (0,2,1,5,4,3,7). \end{aligned}$$

(!) Jeżeli w powyższym przykładzie przejście w głąb nie byłoby zatrzymywane po napotkaniu stanu 7 (końcowego), to pełny porządek odwiedzanych stanów miałby postać: $(0,1,3,7,4,5,6)$.

Rekonstrukcję ścieżki we wszystkich algorytmach grafowych można zrealizować poprzez wsteczne przejście po wskaźnikach na rodzica, poczynając od stanu końcowego zwróconego jako rozwiązanie.

Na poziomie programistycznym, zbiór *Open* w algorytmie Dijkstry jest powszechnie implementowany jako *kolejka priorytetowa* (ang. *priority queue*). Jest to struktura danych oparta najczęściej na *kopcu binarnym* (ang. *binary heap*), pozwalająca na wydajne pobieranie kolejnych stanów z zachowaniem porządku nałożonego przez funkcję g. Takie pobranie wraz z usunięciem elementu (stanu) z głowy kolejki[2] ma logarytmiczną złożoność obliczeniową — $O(\log n)$, gdzie n oznacza liczbę wszystkich elementów. Dodanie nowego elementu do kolejki priorytetowej ma również złożoność $O(\log n)$ przy uwzględnieniu amortyzacji[3].

2.2 Czy znamy rozmiar grafu z góry?

Bardzo istotny wpływ na złożoność algorytmu Dijkstry oraz kolejnych algorytmów, które omówimy w następnych sekcjach, ma fakt, czy rozmiar grafu jest znany z góry (tj. czy znane jest n). Jeżeli rzeczywiście tak jest, to wiele kroków algorytmu można zrealizować w sposób tani obliczeniowo dzięki implementacji tablicowej. Stany mogą zostać wówczas po prostu ponumerowane liczbami całkowitymi, a przechowane w tablicach mogą być:

- wartości funkcji g,
- flagi odwiedzin w zbiorze *Closed*,
- oraz pomocniczo pozycje stanów w zbiorze *Open* (kolejce priorytetowej).

Warto zauważyć, patrząc na algorytm Dijkstry, że warunki: $t \in Closed$ (linia nr 11), $t \notin Open$ (linia nr 14) oraz warunek poprawy kosztu (linia nr 17) można dzięki implementacji tablicowej sprawdzić w czasie stałym — $O(1)$. Ponadto, krok podmiany stanu w zbiorze *Open* nowym egzemplarzem (linia nr 18) można zrealizować w czasie $O(\log n)$[4].

Niestety, w większości zadań rozaptrywanych w ramach sztucznej inteligencji i „atakowanych" podejściem grafowym, **rozmiar przeszukiwanego grafu nie jest znany z góry**. W problemach takich jak np. sudoku, kostka Rubika, puzzle przesuwne, upakowania prostokątne czy choćby nawigowanie (dla odpowiednio dużej mapy) graf jest po prostu eksplorowany przez algorytm w sposób dynamiczny. Przeszukiwanie rozpoczyna się znając tylko stan początkowy, a kolejne stany są

[2] tzw. operacja *poll* na kopcu.

[3] Chodzi tu o uwzględnienie momentów, w których dochodzi do rozszerzania się tablicy z kopcem. Rozszerzanie ma koszt liniowy — $O(n)$ — ale następuje z częstością wygaszaną wykładniczo, co w zamortyzowanym przeliczeniu na jedną operację dodania oznacza koszt stały.

[4] Samo przypisanie ma koszt $O(1)$, ponieważ znamy pozycję w kolejce, po czym należy naprawić kopiec operacją *heap up* — koszt $O(\log n)$.

poznawane stopniowo i wynikają z procedury generowania potomków względem bieżącego stanu. Co więcej, nawet ewentualna znajomość dokładnej liczby stanów w pełnym grafie mogłaby okazać się nieprzydatna, jako że liczby te są zwykle astronomicznych rzędów. Dla przykładu liczba unikalnych rozwiązań sudoku dla planszy 9×9 wynosi

$$6\,670\,903\,752\,021\,072\,936\,960 \approx 6.7 \cdot 10^{21}. \tag{2.1}$$

Program rozwiązujący sudoku musi zapewniać możliwość ewidencjonowania różnych cząstkowych wypełnień planszy, stąd też należy wziąć pod uwagę następującą liczbę

$$\sum_{k=0}^{81} \binom{81}{k} = 2^{81} = 2\,417\,851\,639\,229\,258\,349\,412\,352 \approx 2.4 \cdot 10^{24}. \tag{2.2}$$

czyli liczbę możliwych podzbiorów zbioru 81-elementowego.

A zatem podejścia grafowe dla zadań stawianych w ramach sztucznej inteligencji przeszukują zwykle tylko pewien mały podzbiór całej przestrzeni przeszukiwań. Oznacza to również, że implementacje z wykorzystaniem prostego tablicowania dla wymienionych wcześniej elementów nie są możliwe. W konsekwencji przyjmuje się zwykle, że:

- wartości funkcji g (lub innych funkcji kosztu) są przechowywane wewnątrz samych stanów (jako własność / pole obiektu),
- zbiór *Closed* implementuje się z użyciem *tablicy mieszającej*[5] (ang. *hash table*),
- nie zapamiętuje się pozycji stanów w zbiorze *Open* pozostając przy standardowej kolejce priorytetowej (co prowadzi do liniowych kosztów sprawdzenia, obecności oraz podmiany stanu)
- lub też buduje się nową strukturę danych dla zbioru *Open* rozszerzając kolejkę priorytetową o pomocniczą tablicę mieszającą kosztem pamięci (aby uniknąć wspomnianej liniowej złożoności obliczeniowej).

Zastosowanie tablic mieszających zachowuje złożoność stałą — $O(1)$ — najistotniejszych operacji[6], przy czym programista musi zdefiniować sposób obliczania wartości funkcji mieszającej dla stanów. Takie obliczenia mogą odbywać się na podstawie zawartości pól wewnątrz obiektu stan (i pewnej konkatenacji tych pól) lub na podstawie pewnej unikalnej napisowej reprezentacji obiektu stan[7].

[5] zwanej także *mapą mieszającą* (ang. *hash map*).

[6] Dokładniej: sprawdzenie obecności stanu — koszt $O(1)$ w najgorszym przypadku, dodanie nowego stanu — zamortyzowany koszt $O(1)$ oraz $O(n)$ w najgorszym przypadku.

[7] Np. w języku Java stosuje się w tym zakresie zwykle metody `toString()` oraz `hashCode()`.

2.3 Przeszukiwanie poinformowane

2.3.1 Best-first search

W pracy [Pea84] Judea Pearl zaproponował takie podejście do przeszukiwania, w którym algorytm w pierwszej kolejności odwiedza i rozwija dalej zawsze najbardziej obiecujący — *najlepszy* — stan (ang. *best-first*).

Ocena, na ile dany stan s jest obiecujący, odbywa się poprzez pewną ***funkcję heurystyczną***, którą będziemy oznaczać przez $h(s)$. Funkcja ta może być konstruowana na różne sposoby. W ogólności wartości zwracane przez nią mogą zależeć od:

- informacji zawartej w samym stanie s (statyczny opis s),
- informacji zebranej wzdłuż ścieżki przebytej aż do s,
- ogólnej wiedzy o problemie,
- własności stanu będącego rozwiązaniem.

Dokładne kroki przeszukiwania za pomocą podejścia best-first prezentuje pseudokod oznaczony jako Algorytm 4.

Algorytm 4 Best-first search

```
1:  procedura BestFirstSearch(s_0)                          ▷ stan początkowy: s_0
2:      Closed := ∅                                         ▷ pusty zbiór odwiedzonych stanów
3:      oblicz h(s_0)                                       ▷ heurystyka wg podanego przepisu
4:      ustaw pusty wskaźnik z s_0 na jego rodzica
5:      Open := {s_0}                                       ▷ kolejka stanów oczekujących
6:      dopóki Open ≠ ∅ wykonaj
7:          pobierz (i usuń) z Open stan s o najmniejszej wartości h(s)   ▷ operacja „poll”
8:          jeżeli s jest stanem końcowym to zwróć s        ▷ znaleziono rozwiązanie
9:          wygeneruj zbiór stanów {t} potomnych dla s
10:         dla wszystkich t wykonaj
11:             jeżeli t ∈ Closed to kontynuuj od kolejnej iteracji   ▷ t już odwiedzone
12:             oblicz h(t)
13:             ustaw wskaźnik rodzica t na s
14:             jeżeli t ∉ Open to
15:                 dodaj t do Open
16:             w przeciwnym razie
17:                 jeżeli nowa wartość h(t) jest mniejsza niż znana dotychczas to
18:                     zastąp t w Open nowym egzemplarzem (aktualnie badanym)
19:                     uaktualnij pozycję t w Open
20:         dodaj s do Closed
21:     zwróć wynik pusty                                  ▷ nie znaleziono rozwiązania
```

Zwyczajowo w przeszukiwaniu grafów przyjmuje się, że $h(s)$ jest funkcją o wartościach nieujemnych ($h(s) \geqslant 0$), a wartości bliskie zeru sugerują bliskość stanu s do rozwiązania (do stanu docelowego). Nieformalnie można zatem trakować $h(s)$

jako funkcję odległości, chociaż w sensie ścisłym nie musi ona spełniać własności metryki.

Pewnego komentarza wymaga fragment algorytmu zawartych w liniach o numerach 16–19. W tym fragmencie algorytm wykrywa, że dla pewnego potomka t jego inny egzemplarz istnieje już w zbiorze *Open*, w związku z tym algorytm sprawdza, czy nowo obliczona wartość $h(t)$ jest mniejsza (lepsza) niż wartość znana dotychczas. Można zadać tu pytanie: czy funkcja heurystyczna może zwracać różne wartości dla dwóch egzemplarzy tego samego stanu? W ogólności odpowiedź na to pytanie jest twierdząca, i ma to miejsce np. gdy funkcja h nie jest wyłącznie funkcją samego opisu stanu s, a bierze pod uwagę także np. informacje zebrane wzdłuż ścieżki od s_0 do s. Należy jednak zaznaczyć, że takie przypadki należą do rzadkości w praktyce i w wielu powszechnych zastosowaniach algorytmu best-first search funkcja h jest projektowana jako funkcja stała dla różnych egzemplarzy tego samego stanu.

Przeszukiwania podejściem best-first i heurystyki w nich stosowane są zwykle zaprojektowane tak, aby osiągnąć stan docelowy szybko, za pomocą *dowolnej* ścieżki. Innymi słowy, algorytm best-first search nie koncentruje się na jakiejkolwiek optymalizacji ścieżki — nie dba o liczbę wykonanych manipulacji ani o koszt ścieżki. Tak naprawdę funkcja kosztu ścieżki przebytej (o znaczeniu równoważnym do funkcji g w algorytmie Dijkstry) nie istnieje w algorytmie best-first search.

2.3.2 Przykłady heurystyk dla „puzzli przesuwnych”

Puzzle przesuwne (ang. *sliding puzzle*) to jednoosobowa układanka rekreacyjna wymyślona przez Noyesa Chapmana w 1880 r. Gracz otrzymuje planszę z płaskimi elementami — kafelkami, na które naniesiony jest pewien obrazek lub cyfry. Kafelki są rozłożone na siatce prostokątnej, najczęściej kwadratowej, przy czym jeden z nich jest odjęty stwarzając puste miejsce, co pozwala na przesuwanie kafelków sąsiednich. W stanie początkowym kafelki są pomieszane, a zadaniem gracza jest wykonanie takich przesunięć, które odtworzą oryginalny układ (obrazek lub porządek cyfr).

W przypadku kwadratowej siatki $n \times n$ kafelków, puzzle przesuwne bywają także nazywane układanką $n^2 - 1$ (np. układanka 8, układanka 15), z uwagi na odjęty kafelek służący do przesunięć, patrz rys. 2.2. Układanki 8-elementowe są stosunkowo łatwo rozwiązywalne przez dzieci. Z kolei niektóre układanki 15-elementowe mogą być bardzo trudne dla dorosłych i wymagać nawet przynajmniej 50 ruchów.

Dodatkowo warto zwrócić uwagę, że w „ludzkiej” wersji tej układanki, podobnie jak w przypadku kostki Rubika, nie wymaga się od gracza, aby znalazł minimalną sekwencję przesunięć prowadzącą do rozwiązania. Tego typu wymóg

Rys. 2.2: Plansze układanki puzzle przesuwne (źródło: *Google Images*).

można z kolei nałożyć na program komputerowy rozwiązujący puzzle przesuwne[8]. W zależności od tego, czy wymóg ten obowiązuje czy nie, puzzle przesuwne można rozwiązywać odpowiednio algorytmem best-first search lub A*, wykorzystując w obu przypadkach pewną funkcję heurystyczną. Poniżej przedstawiamy przykłady trzech takich funkcji [HMY85].

Heurystyka „kafelki na niewłaściwych miejscach" (ang. *„misplaced tiles"*)

Zgodnie ze swoją nazwą ta funkcja heurystyczna zwraca dla danego stanu s (czyli dla planszy układanki reprezentowanej przez s) liczbę kafelków przebywających na niewłaściwym miejscu, przy czym w zliczaniu nie bierze udziału kafelek pusty oznaczony zwykle numerem 0. Powyższy sens jest na tyle prosty, że nie wymaga on żadnego wzoru matematycznego, niemniej jednak przedstawiamy takowy poniżej, między innymi w celu ułatwienia zrozumienia kolejnych heurystyk.

Niech $s(i,j)$ oznacza numer kafelka stojącego na przecięciu i-tego wiersza i j-tej kolumny (numerowanie od zera: $i,j = 0,1,\ldots,n-1$). Przyjmujemy, że prawidłowo ułożona plansza (stan docelowy) czytana kolejno wierszami od góry do dołu i od lewej do prawej przedstawia numery w porządku: $0,1,\ldots,n^2-1$. Wzór heurystyki „misplaced tiles" zapisany jest poniżej jako suma jedynek z użyciem funkcji wskaźnikowej (notacja $[\cdot]$ oznacza funkcję wskaźnikową zwracającą 1, gdy zdanie będące jej argumentem jest prawdziwe, a 0 w przeciwnym razie):

$$h(s) = \sum_{\substack{0 \leqslant i,j < n \\ s(i,j) \neq 0}} \sum [s(i,j) \neq in + j]. \tag{2.3}$$

[8] Wprowadzenie tego dodatkowego wymogu powoduje, że wskazanie najkrótszej ścieżki jest w praktyce poza zasięgiem człowieka.

Heurystyka „Manhattan”

Zamiast doliczać $+1$ za każdy kafelek na niewłaściwym miejscu, można użyć dokładniejszej informacji opartej na odległości takiego kafelka od jego miejsca docelowego. Ze względu na wykonywanie ruchów tylko wzdłuż osi pionowej lub poziomej właściwa tu jest metryka Manhattan (nazywana także odległością miejską lub taksówkową). Wiadomo przecież, że każdy kafelek na niewłaściwym miejscu musi pokonać drogę równą przynajmniej odległości Manhattan do właściwego miejsca.

Traktując lewy górny narożnik planszy jako punkt $(0,0)$, można łatwo obliczyć współrzędne docelowe dla kafelka dowolnego numeru k z użyciem dzielenia całkowitego i reszty. Współrzędne te są równe $(\lfloor k/n \rfloor, k \bmod n)$, np. dla $k=7$ i $n=3$ otrzymujemy $(\lfloor 7/3 \rfloor, 7 \bmod 3) = (2,1)$ — 7-ka powinna leżeć w wierszu nr 2 i kolumnie nr 1 (numerowanie od zera). A zatem wzór heurystyki „Manhattan” można zapisać następująco:

$$h(s) = \sum_{\substack{0 \leqslant i,j < n \\ s(i,j) \neq 0}} \sum \left(|i - \lfloor s(i,j)/n \rfloor| + |j - s(i,j) \bmod n| \right), \tag{2.4}$$

co wyraża sumę odległości Manhattan poszczególnych kafelków od ich miejsc docelowych (ponownie z pominięciem kafelka nr 0).

Heurystyka „Manhattan + konflikty liniowe” (ang. *„Manhattan + linear conflicts”*)

Hanson, Mayer i Yung — autorzy pracy [HMY85] poświęconej konstruowaniu heurystyk, zaobserwowali, że w ocenie odległości danego stanu puzzli przesuwnych od rozwiązania ważną rolę odgrywają tzw. *konflikty liniowe*. Zaproponowana przez nich heurystyka zawiera (oprócz standardowego składnika Manhattan) także liczbę konfliktów liniowych pomnożoną przez 2.

Czym jest konflikt liniowy? Przypuśćmy, że górny wiersz układanki 15-elementowej ma postać: $1,2,3,0$. Kafelki w tym wierszu nie są na swoim miejscu, a suma odległości Manhattan wynosi 3. W tym przykładzie nie występują żadne konflikty liniowe, ponieważ przesuwając kafelek 0 trzykrotnie w lewo (lub równoważnie: przesuwając lewego sąsiada kafelka 0 na jego miejsce) otrzymujemy prawidłowe ułożenie kafelków w tym wierszu. Rozważmy teraz inny układ w górnym wierszu: $2,1,3,0$. Ponownie suma odelgłości Manhattan wynosi 3, jednak w tym przykładzie występuje konlikt liniowy ponieważ kafelek 1 leży na prawo od kafelka 2. Innymi słowy, mniejszy numer następuje po większym. W konsekwencji takiego układu podczas rozwiązywania jeden z kafelków będących w konflikcie będzie musiał (prędzej czy później) być przesuniętym do sąsiedniego wiersza i po pewnym czasie (m.in. po naprawieniu konfliktu) być przesuniętym

powrotnie do właściwego wiersza. Z tego właśnie powodu każdy konflikt liniowy wymaga przynajmniej dwóch dodatkowych ruchów.

Konflikty liniowe powinny być zliczane zarówno w wierszach jak i w kolumnach, przy czym muszą być zliczane starannie. Po pierwsze, konflikt liniowy może zachodzić tylko pomiedzy numerami, które przebywają we właściwym wierszu (lub właściwej kolumnie). Np. w górnym wierszu postaci $2,7,3,0$ nie zachodzi konflikt liniowy pomiędzy 7 a 3, pomimo że $7>3$, ponieważ ten wiersz nie jest właściwym dla kafelka 7. Po drugie, konflików nie należy zliczać nadmiarowo. Rozważmy dwa przykładowe układy dla wiersza numer jeden (drugi wiersz od góry), który powinien zawierać numery: $4,5,6,7$. Układ: $7,4,5,6$ w tym wierszu ma tak naprawdę jeden konflikt liniowy, pomimo że $7>4$, $7>5$, i $7>6$. Wystarcza bowiem przesunięcie kafelka 7 do sąsiedniego wiersza i wprowadzenie powrotne za kafelkiem 6 (po odpowiednich manipulacjach z wykorzystaniem kafelka 0). Stąd też układ: $7,6,5,4$ ma trzy konflikty liniowe (7 ma konflikt z czymkolwiek na prawo, 6 z czymkolwiek na prawo, i 5 z 4).

Aby podać ostateczny wzór funkcji heurystycznej, zdefiniujmy zatem najpierw ściśle zbiory konliktów liniowych dla poszczególnych wierszy i kolumn. Niech R_i oznacza zbiór konfliktów liniowych w wierszu i, zaś C_j zbiór konfliktów liniowych w kolumnie j:

$$\begin{aligned} R_i(s) &= \{(i,j)\colon \lfloor s(i,j)/n\rfloor = i,\ \exists_{k>j}\ \text{t.ż. } \lfloor s(i,k)/n\rfloor = i,\ s(i,j) > s(i,k)\},\\ C_j(s) &= \{(i,j)\colon s(i,j) \bmod n = j,\ \exists_{k>i}\ \text{t.ż. } s(k,j) \bmod n = j,\ s(i,j) > s(k,j)\}. \end{aligned} \tag{2.5}$$

Teraz funkcja heurystyczna o nazwie „Manhattan + konflikty liniowe” może zostać wyrażona wzorem

$$\begin{aligned} h(s) = &\sum_{\substack{0\leqslant i,j<n\\ s(i,j)\neq 0}}\sum \Big(|i-\lfloor s(i,j)/n\rfloor| + |j - s(i,j) \bmod n|\Big)\\ &+2\left(\sum_{0\leqslant i<n} \#R_i(s) + \sum_{0\leqslant j<n} \#C_j(s)\right), \end{aligned} \tag{2.6}$$

gdzie znak # oznacza moc zbioru.

2.3.3 Przykłady heurystyk dla sudoku

Sudoku to łamigłówka polegająca na uzupełnianiu cyfr w komórkach pewnej planszy. W najbardziej powszechnej wersji plansza sudoku jest kwadratową siatką komórek o wymiarach 9×9, w której wyróżnione jest 9 wewnętrznych podkwadratów, każdy rozmiarów 3×3. W stanie początkowym plansza jest tylko częściowo wypełniona cyframi. Zadaniem osoby rozwiązującej jest uzupełnić brakujące cyfry w taki sposób, aby wszystkie wiersze, kolumny i podkwadraty zawierały wszystkie

cyfry ze zbioru $\{1,2,\ldots,9\}$. Powszechnie uznaje się, że prawidłowo sformułowane sudoku powinno być jednoznaczne, tzn. prowadzić do dokładnie jednego rozwiązania.

Sudoku zostało spopularyzowane przez japońską firmę Nikoli w późnych latach 80. Niektóre źródła [Sho05; Wik19] sugerują, że łamigłówkę tę oryginalnie i anonimowo zaproponował Howard Garns (amerykański projektant układanek z Indiany), i opublikował po raz pierwszy 1979 r. w Dell Magazines pod nazwą *Number Place*.

Uogólnione na większe rozmiary sudoku można zdefiniować następująco. Dana jest plansza $n^2 \times n^2$ komórek, zawierająca n^2 podkwadratów (każdy wymiarów $n \times n$). Plansza jest częściowo wypełniona cyframi. Celem jest uzupełnienie brakujących cyfr w taki sposób, aby wszystkie wiersze, kolumny i podkwadraty zawierały wszystkie cyfry ze zbioru $\{1,2,\ldots,n^2\}$. Tym samym tradycyjne sudoku odpowiada przypadkowi $n = 3$. Dla $n = 4$ i $n = 5$ otrzymujemy większe sudoku, odpowiednio 16×16 i 25×25. Z kolei $n = 2$ zadaje sudoku 4×4 przeznaczone dla dzieci.

W implementacji komputerowej plansza sudoku może być reprezentowana jako dwuwymiarowa tablica, a niewiadome w poszczególnych komórkach jako wartości 0. W kolejnych akapitach będziemy utożsamiać $s(i,j) \in \{0,1,\ldots,n^2\}$ z wartościami takiej właśnie tablicy. Dwoma ważnymi elementami, które musi dobrać projektant programu do rozwiązywania sudoku, są wybór funkcji heurystycznej oraz wybór sposobu generowania stanów potomnych. Warto zwrócić uwagę, że ten drugi element nie jest zagadnieniem w przypadku układanki puzzle przesuwne — mechanika gry wymusza tam konkretny sposób generowania potomków. W przypadku sudoku teoretycznie można wybrać dowolną komórkę zawierającą niewiadomą, tj. (i,j) takie, że $s(i,j) = 0$ i zamieniać 0 w kolejne numery ze zbioru $\{1,2,\ldots,n^2\}$, generując tym samym stany potomne, o ile tylko nie prowadzi to do natychmiastowej sprzeczności. Można domyślać się jednak (a także sprawdzić eksperymentalnie w ramach ćwiczeń laboratoryjnych), że wybór miejsca do „podpięcia" stanów potomnych będzie wpływał na czasochłonność procesu przeszukiwaniu.

Heurystyka „liczba niewiadomych"

Bardzo prostą (lub wręcz prymitywną) heurystykę możemy zdefiniować następująco:

$$h(s) = \sum_{0 \leqslant i,j < n^2} \sum [s(i,j) = 0]. \tag{2.7}$$

Poszukiwanie rozwiązania sudoku z użyciem algorytmu best-first search oraz powyższej heurystyki zdegeneruje tak naprawdę cały proces do algorytmu depth-first search. Niemniej, odpowiedni wybór komórki z niewiadomą w celu generowania

potomków może (pomimo prostoty tej heurystyki) i tak skutecznie ograniczać błądzenie i prowadzić algorytm stosunkowo szybko do rozwiązania.

Heurystyka „suma pozostałych możliwości"

Intuicyjnie dobrym pomysłem wydaje się odwiedzanie w pierwszej kolejności tych stanów sudoku, dla których łączna liczba pozostałych możliwości wzięta po wszystkich komórkach z niewiadomymi jest najmniejsza. Pomysł ten można przenieść na funkcję heurystyczną. Niech $R_{i,j}(s)$ oznacza zbiór pozostałych możliwych cyfr do wspisania w komórkę $s(i,j)$ po odjęciu ze zbioru $\{1,\ldots,n^2\}$ cyfr już występujących w wierszu i, kolumnie j oraz podkwadracie, który zawiera komórkę (i,j). Heurystykę można wówczas określić następująco:

$$h(s) = \sum_{0 \leqslant i,j < n^2} \sum \#R_{i,j}(s). \tag{2.8}$$

Przykłady działania algorytmu best-first search w procesie rozwiązywania sudoku z użyciem przedstawionych powyżej dwóch heurystyk znajdują się w punkcie 2.3.5.

2.3.4 A*

Algorytm A* zaproponowali Haart, Nilsson i Raphael w pracach [HNR68; HNR72]. Nieformalnie algorytm ten może być rozumiany jako połączenie algorytmów Dijkstry i best-first search (lub jako ich ogólniejszy wariant). Jest tak, ponieważ A* używa zarówno dokładnego kosztu g drogi przebytej jak i heurystycznego kosztu h.

Uściślając, funkcja oceny decydująca o porządku pobierania stanów ze zbioru *Open*, ma w algorytmie A* postać:

$$f(s) = g(s) + h(s), \tag{2.9}$$

gdzie $g(s)$ jest dokładnym kosztem (lub odległością) zaobserwowanym podróżując od stanu początkowego s_0 aż do stanu s, a $h(s)$ jest heurystycznym oszacowaniem kosztu pozostałego od stanu s do stanu docelowego. Jako że $h(s)$ jest składnikiem heurystycznym, to cała funkcja $f(s)$ może być również traktowana jako heurystyczna. Pseudokod oznaczony jako Algorytm 5 prezentuje dokładne kroki algorytmu A*.

W zastosowaniach związanych z poszukiwaniem najkrótszej ścieżki (tj. ścieżki o najmniejszym koszcie) kluczowym jest, aby funkcja h była tzw. ***heurystyką dopuszczalną*** (ang. *admissible heuristics*). Oznacza to, że funkcja h nie może przeszacowywać nieznanego prawdziwego kosztu pozostałego do celu.

Algorytm 5 A*

```
1:  procedura AStar(s_0)                                   ▷ stan początkowy: s_0
2:      Closed := ∅                                        ▷ pusty zbiór odwiedzonych stanów
3:      g(s_0) := 0                                        ▷ koszt przebyty od startu
4:      oblicz h(s_0)                                      ▷ heurystyka wg podanego przepisu
5:      f(s_0) := g(s_0) + h(s_0)                          ▷ suma decydująca o porządku pobierania z Open
6:      ustaw pusty wskaźnik rodzica dla s_0
7:      Open := {s_0}                                      ▷ kolejka stanów oczekujących
8:      dopóki Open ≠ ∅ wykonaj
9:          pobierz (i usuń) z Open stan s o najmniejszej wartości f(s)   ▷ operacja „poll"
10:         jeżeli s jest stanem końcowym to zwróć s       ▷ znaleziono rozwiązanie
11:         wygeneruj zbiór stanów {t} potomnych dla s
12:         dla wszystkich t wykonaj
13:             jeżeli t ∈ Closed to kontynuuj od kolejnej iteracji   ▷ t już odwiedzone
14:             g(t) := g(s) + Δ(s → t)
15:             oblicz h(t)
16:             f(t) := g(t) + h(t)
17:             ustaw wskaźnik rodzica t na s
18:             jeżeli t ∉ Open to
19:                 dodaj t do Open
20:             w przeciwnym razie
21:                 jeżeli nowa wartość f(t) jest mniejsza niż poprzednio znana to
22:                     zastąp t w Open nowym egzemplarzem (aktualnie badanym)
23:                     uaktualnij pozycję t w Open
24:         dodaj s do Closed
25:     zwróć wynik pusty                                 ▷ nie znaleziono rozwiązania
```

Formalna definicja heurystyki dopuszczalnej jest następująca:

Definicja 2.3.1 — heurystyka dopuszczalna. Niech $h^*(s)$ oznacza funkcję zwracającą dla każdego stanu s dokładny koszt pozostały od s do stanu docelowego s^*. Mówimy, że heurystyka h jest dopuszczalna wtedy, i tylko wtedy, gdy: $\forall s\ h(s) \leqslant h(s^*)$.

Oczywiście funkcja h^* jest w praktyce nieznana, ale istnieje. Gdybyśmy znali funkcję h^*, to racjonalnym byłoby używanie właśnie jej zamiast innej funkcji h.

Twierdzenie 2.3.1 Niech s^* oznacza stan docelowy, a h dowolną heurystykę dopuszczalną. Jeżeli istnieje przynajmniej jedna ścieżka pomiędzy s_0 a s^*, to algorytm A* (Algorytm 5) gwarantuje znalezienie ścieżki najkrótszej — tj. ścieżki o najmniejszej wartości kosztu g.

Dowód. Algorytm A* zatrzymując się (linia nr 10) zwraca pewien egzemplarz stanu s^* o koszcie $g(s^*)$. Oczywiście $h(s^*) = 0$, ponieważ s^* spełnia warunek stopu.

Wiadomo, że zbiór *Open* zachowuje niemalejący względem funkcji f porządek pobierania stanów. Zatem dla wszystkich stanów s przebywających w chwili stopu w zbiorze *Open* spełniony jest warunek: $f(s) \geqslant f(s^*)$. Należy rozważyć dwa przypadki. (1) Jeżeli pewien stan $s \in Open$ spełnia warunek stopu, to $h(s) = 0$ ale $g(s) \geqslant g(s^*)$, ponieważ $f(s) \geqslant f(s^*)$. Innymi słowy, s jest także egzemplarzem stanu końcowego, ale koszt jego ścieżki jest nietańszy niż koszt ścieżki skojarzonej z s^*. (2) Jeżeli s nie spełnia warunku stopu, ale pozwala na dojście do stanu docelowego (istnieje ścieżka z s do s^*) i aktualnie mamy, że $g(s) \leqslant g(s^*)$, to ostateczna ścieżka z wykorzystaniem s nie może być tańsza niż ścieżka stanu s^*, jako że $h(s)$ będące dolnym ograniczeniem na koszt pozostały wskazuje, że $g(s) + h(s) \geqslant g(s^*)$, ponieważ $f(s) \geqslant f(s^*)$. ■

Użytecznym w powyższym konteście jest także pojęcie *heurystyki monotonicznej* (ang. *monotonous heuristics*).

Definicja 2.3.2 — „heurystyka monotoniczna". Mówimy, że heurystyka h jest monotoniczna wtedy i tylko wtedy, gdy dla wszystkich par s,t (gdzie t jest potomkiem s) spełniony jest warunek:

$$f(s) \leqslant f(t), \tag{2.10}$$

co można równoważnie zapisać jako:

$$\begin{aligned} g(s) + h(s) &\leqslant g(t) + h(t), \\ h(s) &\leqslant g(t) - g(s) + h(t), \\ h(s) &\leqslant \Delta(s \to t) + h(t). \end{aligned} \tag{2.11}$$

Ostatnią nierówność można traktować jako swoisty wariant *nierówności trójkąta* i wypowiedzieć następująco. W wyniku dowolnego przejścia $s \to t$, wartość heurystyczna w punkcie s (sprzed przejścia) jest mniejsza lub równa sumie kosztu przejścia $\Delta(s \to t)$ oraz wartości heurystycznej w punkcie t (po przejściu). A nierówność staje się równością tylko w tych przypadkach, gdy przechodzenie do celu odbywa się po linii prostej[9].

Twierdzenie 2.3.2 Jeżeli heurystyka h jest monotoniczna, to jest także dopuszczalna.

Wynikanie w powyższym twierdzeniu nie pracuje w ogólności w przeciwnym kierunku, co powodowałoby równoważność obu pojęć. Tzn. można wskazać przykłady heurystyk, które są dopuszczalne, ale nie są monotoniczne.

[9] Linii prostej w sensie metryki skojarzonej z danym grafem.

Dlaczego monotoniczność heurystyki jest użyteczną własnością? Wyobraźmy sobie, że zetknęliśmy się z pewnym nowym i trudnym problemem optymalizacyjnym, który chcielibyśmy rozwiązać podejściem przeszukiwania grafu. Przypuśćmy, że opracowaliśmy odpowiednią reprezentację, definiując, czym jest stan w naszym problemie oraz w jaki sposób generować stany potomne, a także wpadliśmy na kilka pomysłów opracowania różnych funkcji heurystycznych, kierując się intuicją. Niestety nie mamy pewności, czy nasze heurystyki zagwarantują znalezienie rozwiązania optymalnego, czyli związanego z najkrótszą ścieżką. Jeżeli będziemy umieli udowodnić monotoniczność pewnej heurystyki, to automatycznie będzie to implikowało jej dopuszczalność, a co za tym idzie pewność, że algorytm A* znajdzie dla nas najkrótszą ścieżkę zgodnie z Twierdzeniem 2.3.1.

Pokażemy, jak działa ten mechanizm na przykładzie heurystyki „kafelki na niewłaściwych miejscach" (ang. *misplaced tiles*) w układance „puzzle przesuwne" omówionej w punkcie 2.3.2. Punktem wyjścia przy udowadnianiu monotoniczności heurystyki jest zawsze nierówność (2.11).

Twierdzenie 2.3.3 Heurystyka „kafelki na niewłaściwych miejscach" jest monotoniczna.

Dowód. Nierówność (2.11) musi zachodzić dla wszystkich par: rodzic s, potomek t. Zauważmy, że w wyniku przesunięcia pewnego kafelka (w miejsce puste) ponosimy zawsze koszt 1 ruchu, tj. $\Delta(s \to t) = 1$, zaś wartość funkcji heurystycznej $h(t)$ w stanie potomnym może:

(a) pozostać nie zmieniona względem rodzica, $h(t) = h(s)$, gdy przesuwany kafelek był i jest nadal na niewłaściwym miejscu, lub
(b) zwiększyć się o 1, $h(t) = h(s)+1$, gdy przesuwany kafelek był na właściwym miejscu przed ruchem, lub
(c) zmniejszyć się o 1, $h(t) = h(s)-1$, gdy przesuwany kafelek trafił na właściwe miejsce w wyniku ruchu.

Przypadki (a) i (b) spełniają (2.11) w formie ścisłej nierówności — odpowiednio: $h(s) < 1+h(s)$ oraz $h(s) < 1+h(s)+1$. Zaś przypadek (c) spełnia (2.11) w formie równości: $h(s) = 1+h(s)-1$. ■

Przemyślenie analogicznych dowodów dla heurystyk „Manhattan" oraz „Manhattan + konflikty liniowe" pozostawiamy jako ćwiczenie dla Czytelnika.

Warto nadmienić, że pierwotnie Haart, Nilsson i Raphael zaproponowali nieco inną nazwę dla swojego algorytmu, mianowicie nazwę: A. Intencją było rozróżnienie notacyjne wiążące heurystyki z algorytmami. Na przykład, jeżeli rozważamy dwie heurystyki h i h^*, gdzie gwiazdka wskazuje optymalność heurystyki (czyli jej równość z funkcją dokładnego kosztu), to algorytm używający h^* można również

nazwać optymalnym i oznaczać jako A*. Co więcej, optymalność takiego algorytmu jest dwojaka. Po pierwsze, jest on najbardziej wydajny wśród wszystkich algorytmów A — tzn. odwiedza najmniej stanów. Po drugie, działa on tak samo dobrze lub lepiej niż *wszystkie* inne algorytmy grafowe do znajdowania najkrótszych ścieżek (niekoniecznie z rodziny algorytmów A), które są równie dobrze poinformowane, tj. poinformowane za pomocą funkcji h^*. Pierwotna idea nazewnicza autorów zatarła się i współcześnie używa się nazwy A* niezależnie od użytej heurystyki.

Jak wspomniano wcześniej, algorytmy Dijkstry i best-first search mogą być rozumiane jako szczególne przypadki algorytmu A*. Wymuszenie w algorytmie A* zerowego składnika kosztu przebytego: $\forall s\ g(s) = 0$, redukuje go do algorytmu best-first search. Z kolei wymuszenie zerowej heurystyki: $\forall s\ h(s) = 0$, redukuje algorytm A* do algorytmu Dijkstry (w wariancie z jednym stanem końcowym).

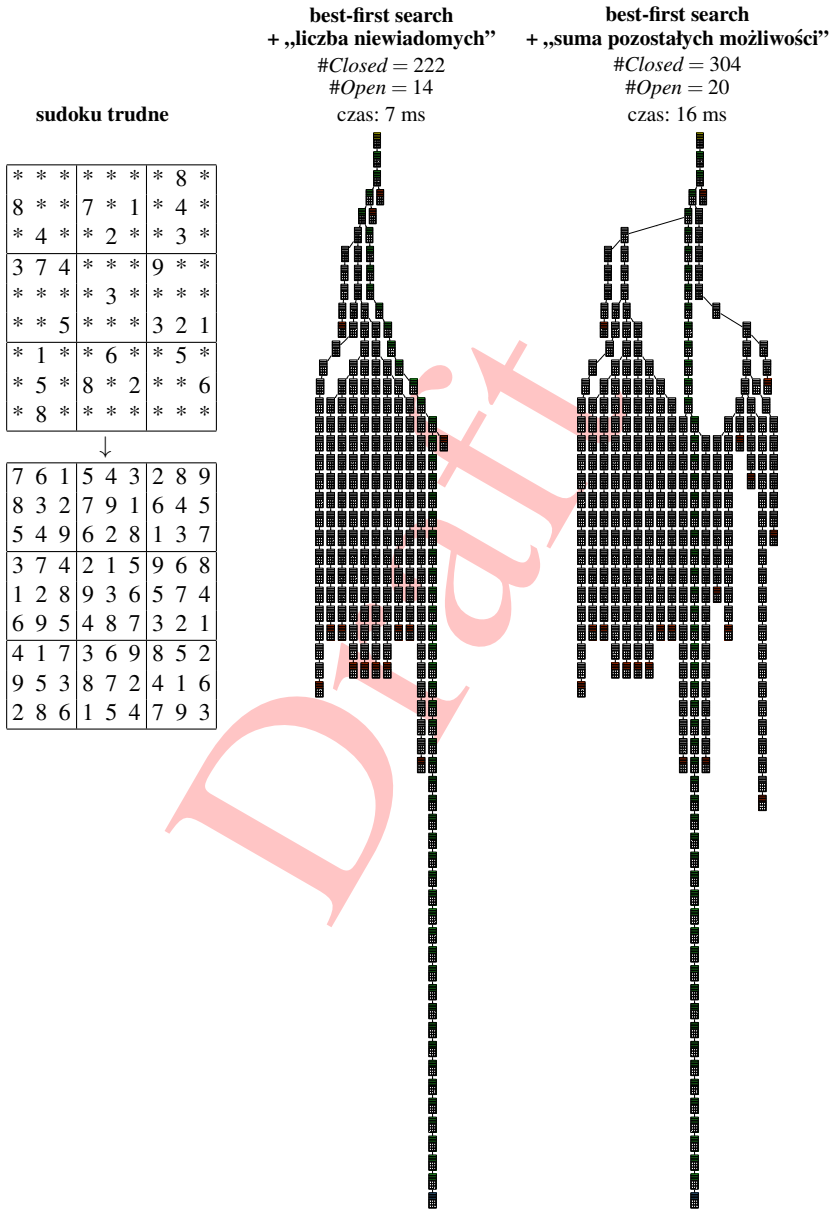
Heurystyka zerowa ($\forall s\ h(s) = 0$) jest w trywialny sposób dopuszczalna.

2.3.5 Przykłady działania best-first search i A*

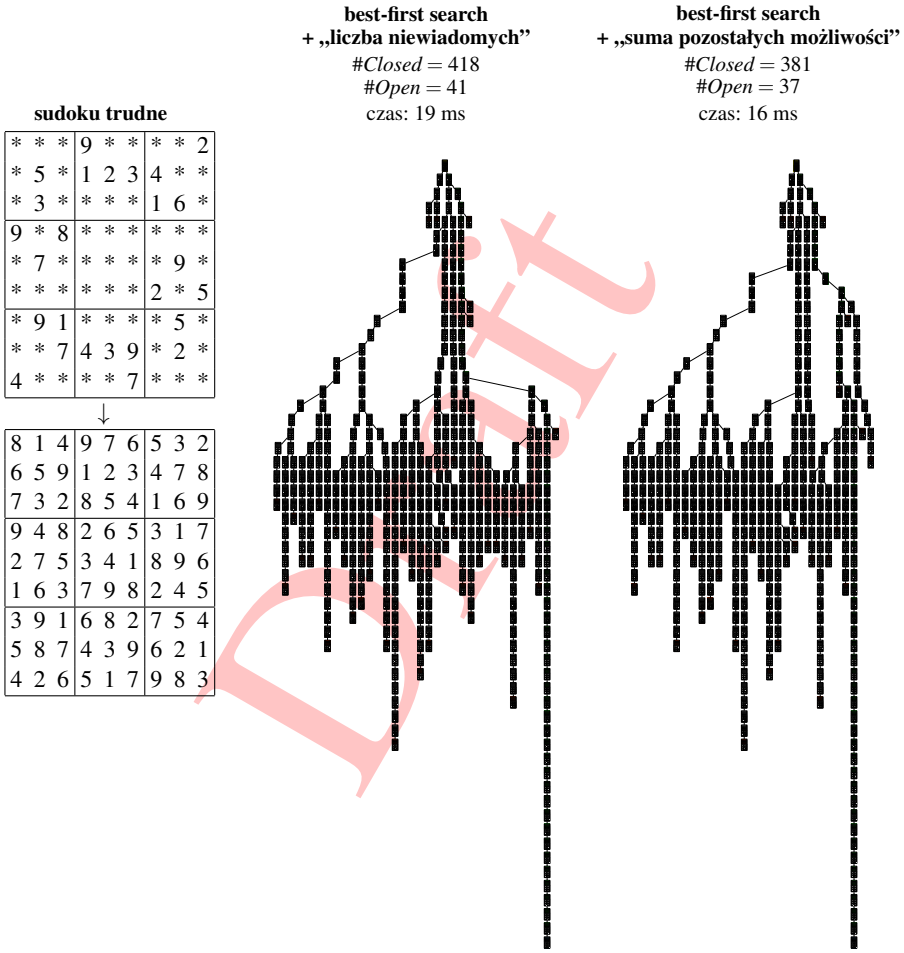
Sudoku

Na rysunkach 2.3, 2.4 przedstawiono grafy przeszukiwań wygenerowane przez algorytm best-first search z użyciem różnych heurystyk dla dwóch łamigłówek sudoku określanych jako trudne. Zachęcamy Czytelnika do powiększenia przeglądanych ilustracji. Kolorem żółtym zaznaczono stan początkowy, a niebieskim końcowy. Kolor szary oznacza stany odwiedzone (w zbiorze *Closed*), a czerwony stany oczekujące (w zbiorze *Open*) w chwili stopu algorytmu. Kolor zielony wskazuje stany będące na ścieżce pomiędzy stanem początkowym a końcowym. Przy każdej ilustracji odnotowano rozmiary zbiorów *Open* i *Closed* w chwili stopu algorytmu, a także czas pracy[10]. We wszystkich przypadkach stany potomne były „zaczepiane" w komórce o namniejszej liczbie pozostałych możliwości (w przypadku remisów — pierwsza napotkana taka komórka idąc od lewego górnego narożnika planszy).

[10]Eksperymenty przeprowadzone na komputerze z procesorem Intel Xeon CPU E3-1505M v5 2.8 GHz (boost 3.7 GHz).

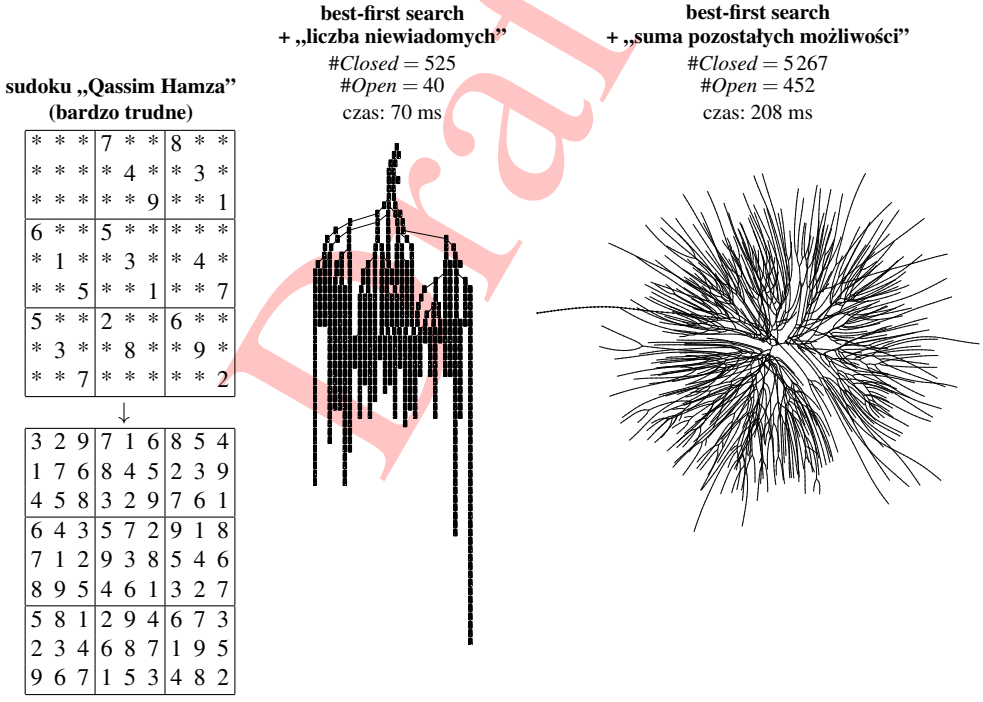


Rys. 2.3: Grafy przeszukiwań dla łamigłówki sudoku (trudne) wygenerowane przez algorytm best-first search z użyciem dwóch różnych heurystyk (źródło: *opracowanie własne*).



Rys. 2.4: Grafy przeszukiwań dla łamigłówki sudoku (trudne) wygenerowane przez algorytm best-first search z użyciem dwóch różnych heurystyk (źródło: *opracowanie własne*).

Interesujący dla sudokistów przykład stanowi sudoku o nazwie „Qassim Hamza”, uważane za przykład bardzo trudny. Dane są 22 wiadome, a trudność wynika z ich ukośnego ułożenia w ramach podkwadratów. Proces rozwiązywania, zarówno rozwiązywania przez człowieka jak i program komputerowy, wymaga w związku z tym dużej liczby zgadnięć (i wycofań z powodu śledzenia błędnych tropów). Rys. 2.5 ilustruje grafy przeszukiwań dla sudoku „Qassim Hamza” wygenerowane podobnie jak poprzednio algorytmem best-first search z użyciem dwóch różnych heurystyk. Tym razem różnica w liczbie odwiedzonych stanów wypada istotnie na korzyść heurystyki „liczba niewiadomych”. Heurystyka „suma pozostałych możliwości” spowodowała zaskakująco dużo błądzenia. Jej graf przeszukiwań zawiera ponad 5 tysięcy stanów i dla czytelności został wyrysowany w sposób promienisty (stan początkowy umieszczony centralnie, kolejne stany potomne oddalają się od środka wraz z głębokością), a stany zilustrowano jako punkty bez zawartości.



Rys. 2.5: Grafy przeszukiwań dla sudoku „Qassim Hamza” (bardzo trudne) wygenerowane przez algorytm best-first search z użyciem dwóch różnych heurystyk (źródło: *opracowanie własne*).

Przykłady z rysunków 2.3–2.5 nie pozwalają jednoznacznie wskazać lepszej z dwóch testowanych heurystyk. Każda z nich opiera się na odmiennym pomyśle i mogą istnieć specyficzne stany początkowe, które będą sprzyjające dla każdej

z nich. Ogólne rozstrzygnięcie, która z heurystyk częściej odwiedza mniejszą liczbę stanów, może być dokonane tylko poprzez odpowiednio duży eksperyment statystyczny (pomiary dla wielu różnych początkowych plansz sudoku).

Przypominamy, że algorytm best-first search nie zwraca uwagi na funkcję kosztu g. Opracowywane heurystyki nie muszą być zatem dopuszczalne ani nawet jakkolwiek powiązane (co do jednostek lub skali) z wartościami funkcji g.

Sztuczny „graf geograficzny"

Rys. 2.6 przedstawia sztucznie stworzony graf przypominający sieć miast i dróg. Graf zawiera 100 wierzchołków (miasta) zamkniętych w kwadracie jednostkowym i około 10% wszystkich możliwych krawędzi (drogi). Koszty krawędzi są proporcjonalne do odległości pomiedzy wierzchołkami w linii prostej z pewnymi losowymi dodatnimi zaburzeniami. Rolę stanu początkowego pełni wierzchołek w lewym górnym narożniku, a końcowego wierzchołek w prawym dolnym narożniku.

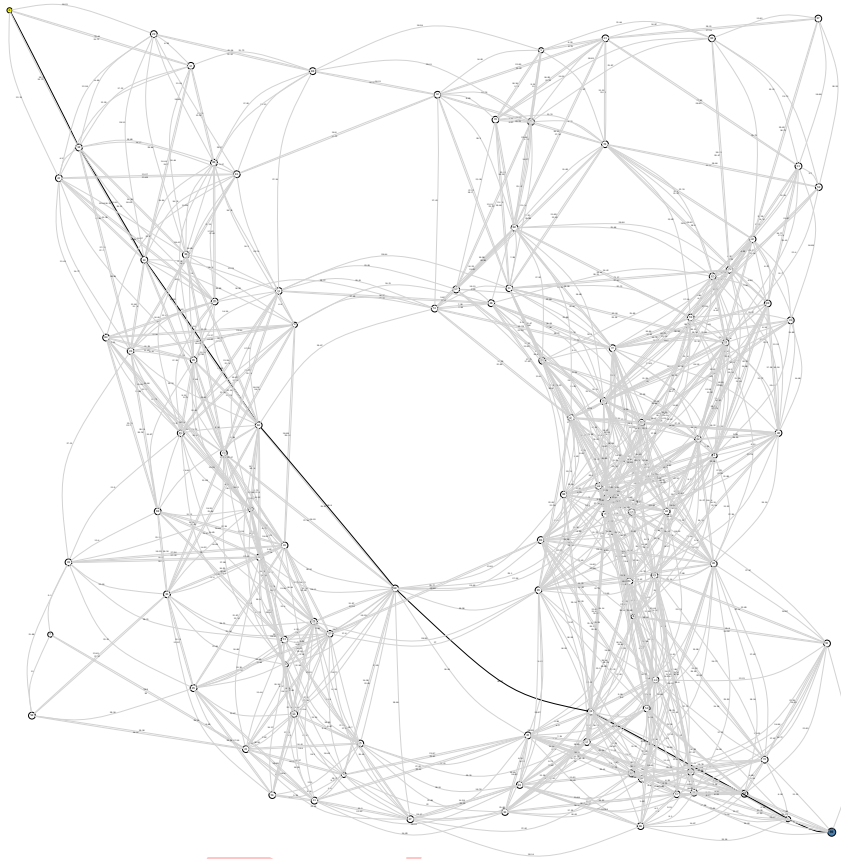
Rysunki 2.7a i 2.7b ilustrują grafy przeszukiwań otrzymane dla sztucznego grafu wejściowego, wygenerowane odpowiednio przez algorytmy Dijkstry i A*. Znając współrzędne geograficzne możemy algorytm A* poinformować tj. wyposażyć go w funkcję heurystyczną obliczającą odległość euklidesową pomiędzy dowolnym stanem a stanem docelowym. Taka informacja znacząco redukuje liczbę odwiedzonych stanów podczas przeszukiwania. Obydwa badane algorytmy znalazły najkrótszą ścieżkę — sekwencję wierzchołków: $(0,18,14,64,60,10,5,99)$ o koszcie ≈ 149.52, przy czym algorytm Dijkstry był zmuszony odwiedzić wszystkie 100 stanów (w związku ze skrajnym położeniem stanów początkowego i końcowego), podczas gdy algorytm A* odwiedził tylko 18 stanów[11].

Puzzle przesuwne

Rys. 2.8 ilustruje przykłady działania algorytmu A* stosującego trzy różne heurystyki dla 8-elementowej układanki puzzle przesuwne o planszy początkowej $(0,3,2;4,7,8;1,5,6)$. W związku z faktem, że każda z badanych heurystyk jest dopuszczalna, algorytm w każdym z trzech warianctów znajduje ścieżkę najkrótszą zawierającą 16 ruchów $(D,R,D,R,U,L,L,D,R,U,U,L,D,R,U,L)$. Jednocześnie można łatwo zauważyć, że najprostsza heurystyka „kafelki na niewłaściwych miejscach" generuje największy graf przeszukiwań (łącznie zbiory *Open* i *Closed* zawierają 672), a kolejne udoskonalone heurystyki „Manhattan" i „Manhattan + konflikty liniowe" istotnie redukują graf przeszukiwań (odpowiednio 106 i 78 stanów łącznie w *Open* i *Closed*).

W odróżnieniu od obserwacji dotyczących algorytmu best-first search i heurystyk dla sudoku (gdzie ciężko było o wskazanie heurystyki lepszej), w przypadku

[11] 18 stanów w zbiorze *Closed*, a 38 w zbiorze *Open* w chwili stopu



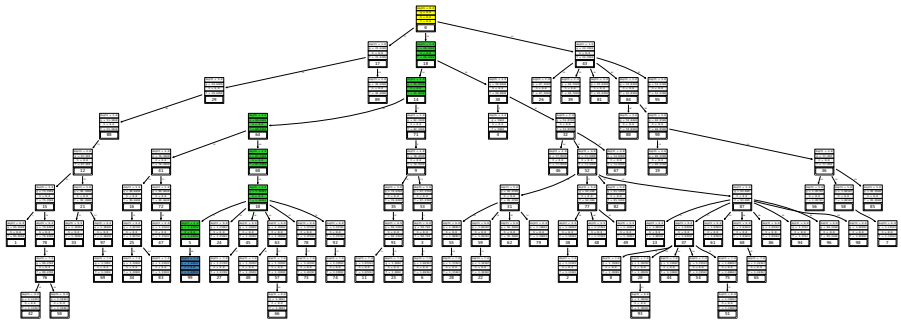
Rys. 2.6: Sztuczny „graf geograficzny" z losowym położeniem 100 wierzchołków w kwadracie jednostkowym (źródło: *opracowanie własne*).

algorytmu A* mamy pewność, że kolejne heurystyki monotoniczne niosące coraz bardziej dokładną informację o prawdziwym koszcie będą regularnie zmniejszały czas pracy algorytmu i jego tendencje do błądzenia. Prawidłowość tę można łatwo zrozumieć w następujący sposób. Niech h_1, h_2, h_3 oznaczają kolejno heurystyki „kafelki na niewłaściwych miejscach", „Manhattan" i „Manhattan + konflikty liniowe". Dla każdego stanu s prawdziwy jest ciąg nierówności

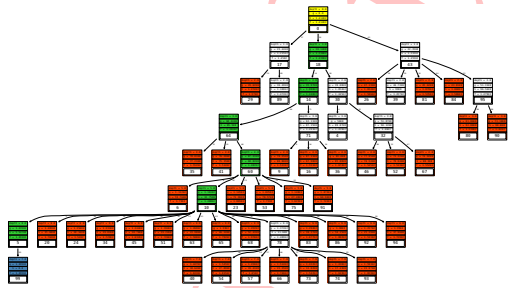
$$0 \leqslant h_1(s) \leqslant h_2(s) \leqslant h_3(s) \leqslant h^*(s), \tag{2.12}$$

gdzie $h^*(s)$ reprezentuje nieznaną funkcję kosztu dokładnego. Lewym skrajem tego ciągu jest wartość 0, którą można utożsamiać z brakiem heurystyki, czyli brakiem informacji nakierowującej na cel (tak jak ma to miejsce w algorytmie Dijkstry). Taka sytuacja oznacza remis pomiędzy wszystkimi stanami o takiej samej

(a) graf przeszukiwań dla grafu z rys. 2.6 wygenerowany przez algorytm Dijkstry



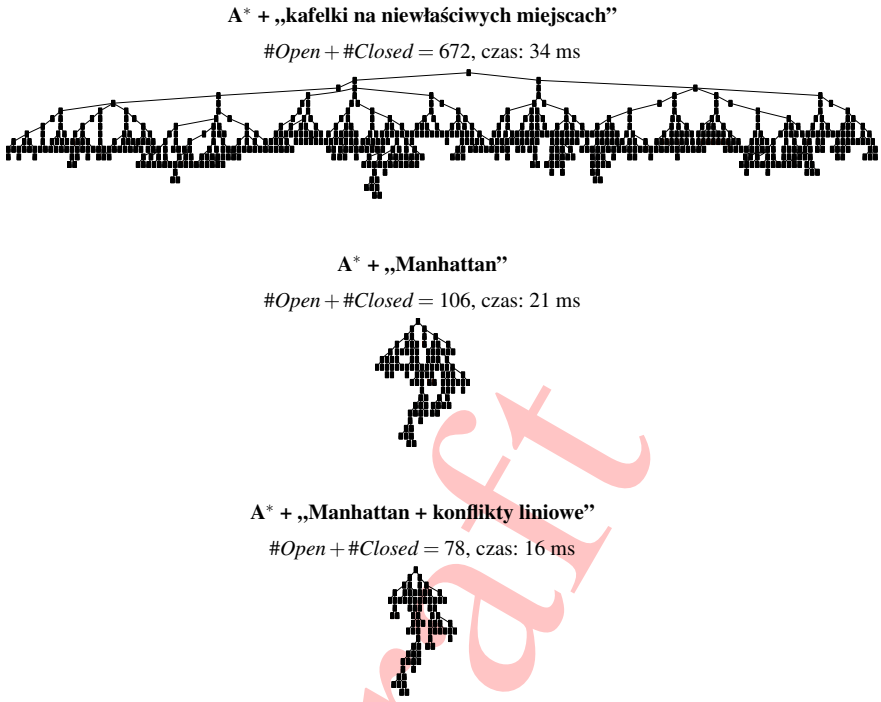
(b) graf przeszukiwań dla grafu z rys. 2.6 wygenerowany przez algorytm A* używający odległości euklidesowej jako heurystyki



Rys. 2.7: Grafy przeszukiwań wygenerowane przez algorytmy Dijkstry i A* dla grafu z rys. 2.6 (źródło: *opracowanie własne*).

wartości kosztu przebytego $g(s)$. Stopniowe przesuwanie się po powyższym ciągu nierówności w prawo jest tożsame z udoskonalaniem informacji nakierowującej na cel. Jednocześnie towarzyszy temu redukcja liczby remisów w ocenie stanów w trakcie pracy algorytmu — innymi słowy sumy $g(s)+h(s)$ rozróżniają coraz lepiej pojawiające się stany potomne z uwagi na używanie heurystyk o coraz większych wartościach (ale nie przeszacowujących kosztu prawdziwego). Prawa skrajność (tj. używanie funkcji h^*) oznacza najmniejszą możliwą liczbę remisów w ocenie stanów. Tak naprawdę remisy mogą się wówczas pojawić tylko w przypadku istnienia dwóch (lub więcej) alternatywnych ścieżek o tym samym minimalnym koszcie.

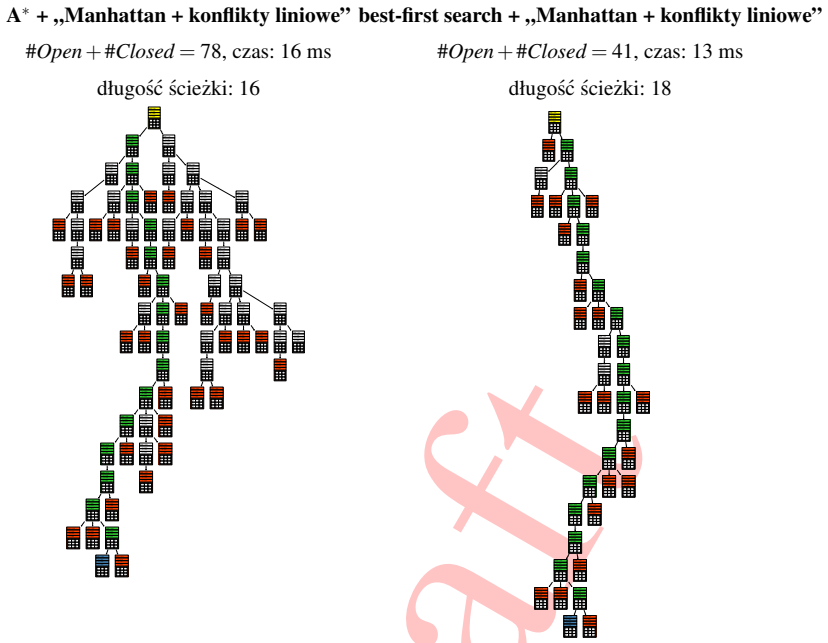
Funkcje heurystyczne o wyższych wartościach (o ile tylko nie przeszacowują prawdziwego kosztu do celu) oznaczają krótszą pracę algorytmu A* — mniejsze tendencje do błądzenia.



Rys. 2.8: Grafy przeszukiwań dla układanki puzzle przesuwne $(0,3,2;4,7,8;1,5,6)$ wygenerowane za pomocą algorytmu A* i trzech różnych heurystyk. (źródło: *opracowanie własne*).

Warto w tym miejscu także zwrócić uwagę na możliwość przeprowadzenia następującego ciekawego eksperymentu. W ogólności dla puzzli n^2-1 istnieje $n!/2$ rozwiązywalnych układów planszy [RN09]. W szczególności dla $n=3$ jest to $9!/2 = 181\,440$ układów. Liczba ta jest na tyle mała, że możliwe jest skonstruowanie funkcji kosztu dokładnego h^* w formie stablicowanej dla $n=3$. Przygotowanie takiej tablicy może być przeprowadzone np. z wykorzystaniem algorytmu A* i dowolnej niedoskonałej funkcji heurystycznej (np. „Manhattan + konflikty liniowe”). Należałoby uruchomić algorytm A* dla każdego z $9!/2$ układów początkowych (oznaczmy taki pojedynczy układ przez s_0), a zaobserwowany wynikowy koszt $g(s^*)$ najkrótszej ścieżki odłożyć do tablicy (np. tablicy mieszającej) — tj. przypisać $h^*(s_0) := g(s^*)$. Tak przygotowana tablica mogłaby służyć jako heurystyka optymalna do ponownego rozwiązywania układanek dla $n=3$ bez błądzenia.

Na koniec tego punktu przedstawiamy rys. 2.9 jako przykład porównania działania algorytmów A* i best-first search uruchomionych dla tej samej układanki puzzli przesuwnych $(0,3,2;4,7,8;1,5,6)$, prezentowanej wcześniej. Jak można



Rys. 2.9: Porównanie działanie algorytmów A* i best-first search dla tej samej układanki „puzzle przesuwne”. (źródło: *opracowanie własne*)

zauważyć algorytm best-first search dociera szybciej do stanu docelowego, odwiedzając mniej stanów, przy czym znajduje ścieżkę *nieoptymalną* złożoną z 18 ruchów: $(R,D,D,R,U,L,U,L,D,D,R,U,U,L,D,R,U,L)$.

(!) Mając do dyspozycji pewną funkcję heurystyczną możemy poszukiwać rozwiązania zarówno algorytmem best-first search jak i A*. Wybór algorytmu powinien być podyktowany tym, czy zależy nam na szybkim dotarciu do rozwiązania jakąkolwiek ścieżką (best-first search) czy też na znalezieniu najkrótszej ścieżki (A*).

(!) Algorytmy A* generują szersze i płytsze grafy przeszukiwań niż algorytmy best-first search dla tych samych problemów.

2.3.6 IDA*

Dla niektórych problemów generowany graf przeszukiwań może być bardzo duży, w związku z czym algorytm A* może napotkać na problemy nadmiernego zużycia pamięci. Liczba stanów w zbiorach *Open* i *Closed* może potencjalnie wyczerpać całą dostępną pamięć RAM.

Algorytm o nazwie Iterative Deepening A* (IDA*)[12] zaproponowany przez Korfa [Kor85] może być postrzegany jako wariant algorytmu A* o małym zużyciu pamięci. IDA* nie przechowuje ewidencji stanów odwiedzonych, tj. nie używa zbioru *Closed*. W trakcie pracy w pamięci przechowywane są tylko stany przebywające na ścieżce, którą aktualnie algorytm bada. W zależności od sposobu implementacji, IDA* może używać małego zbioru *Open* (implementacja nierekurencyjna z użyciem głównej pętli) lub też nie używa wcale zbioru *Open* (implementacja rekurencyjna). Oczywiście, niskie zużycie pamięci nie przychodzi „za darmo" — algorytm IDA* jest wolniejszy niż A*, ponieważ odwiedza wiele stanów wielokrotnie.

Główny pomysł działania IDA* można naszkicować następująco. Na samym początku, algorytm oblicza wartość heurystyki dla stanu początkowego — $h(s_0)$, i tym samym ustala tzw. *horyzont przeszukiwań* $H = f(s_0) = 0 + h(s_0)$. Następnie, wychodząc od stanu początkowego, algorytm podąża różnymi ścieżkami w stylu zbliżonym do przechodzenia techniką depth-first. Jeżeli algorytm napotka na stan końcowy w ramach ustalonego horyzontu (tj. obserwowany koszt g napotkanego stanu jest mniejszy lub równy H), to zatrzymuje się zwracając stan końcowy wraz ze skojarzoną z nim ścieżką. Za każdym razem, gdy algorytm osiąga pewien stan poza horyzontem przeszukiwań, taki stan nie jest rozwijany dalej (jego potomkowie nie sa generowani), przy czym algorytm może wykorzystać koszt zaobserwowany dla tego stanu i jego heurystykę w celu ustalenia nowego horyzontu przeszukiwań H'. Mówiąc ściślej, nowy horyzont przeszukiwań zostaje zdefiniowany jako:

$$H' = \min_{\{s:\, g(s)>H\}} f(s). \tag{2.13}$$

Ostatecznie, gdy wszystkie ścieżki sięgające poza dotychczasowy horyzont H zostaną wyczerpane, to algorytm *pogłębia* horyzont przeszukiwań poprzez przypisanie $H := H'$ i cały proces powtarza się.

Poniżej przedstawiamy dwa pseudokody algorytmu IDA* różniące się sposobem implementacji (rekurencyjna i pętlowa).

W celu zobrazowania zysków i strat związanych z użyciem algorytmu IDA* przedstawiamy tabelę 2.1. Stanowi ona porównanie działania algorytmów IDA* i A* dla pięciu wybranych układanek „puzzle przesuwne" dla przypadku $n = 4$. Układanki wybrano na podstawie informacji z pracy [HMY85] Hanssona, Mayera i Yunga. Przy każdym uruchomieniu pamięć RAM dostępna dla procesu została celowo ograniczona do 2 GB. Wyczerpanie tego limitu powodowało nieprawidłowe zatrzymanie się programu.

Jak można zauważyć, algorytm IDA* zakończył się powodzeniem w każdym z pięciu uruchomień, przy czym niekiedy wiązało się to z bardzo długimi czasami

[12] iteracyjnie pogłębiane A*

Algorytm 6 Rekurencyjny IDA*

```
1:  procedura RecursiveIterativeDeepeningAStar(s_0)        ▷ stan początkowy: s_0
2:      g(s_0) := 0                                        ▷ koszt przebyty od startu
3:      oblicz h(s_0)                                      ▷ heurystyka wg podanego przepisu
4:      f(s_0) := g(s_0) + h(s_0)
5:      ustaw pusty wskaźnik na rodzica dla s_0
6:      H := f(s_0)                                        ▷ początkowy horyzont przeszukiwań
7:      dopóki prawda wykonaj
8:          (s, H') := Search(s_0, H)
9:          jeżeli s ≠ null to zwróć s                     ▷ znaleziono rozwiązanie
10:         jeżeli H' = ∞ to zwróć null                    ▷ nie znaleziono rozwiązania
11:         H := H'
12: procedura Search(s, H)
13:     jeżeli f(s) > H to zwróć (null, f(s))
14:     jeżeli s jest stanem końcowym to zwróć (s, g(s))   ▷ znaleziono rozwiązanie
15:     H' := ∞
16:     wygeneruj zbiór stanów {t} potomnych dla s
17:     dla wszystkich t wykonaj
18:         g(t) := g(s) + Δ(s → t)
19:         f(t) := g(t) + h(t)
20:         (u, H'') := Search(t, H)
21:         jeżeli u ≠ null to zwróć (u, g(u))             ▷ znaleziono rozwiązanie
22:         H' := min{H', H''}                             ▷ pogłębianie horyzontu
        zwróć (null, H')
```

Tabela 2.1: Porównanie działania algorytmów IDA* i A* dla wybranych układanek puzzle przesuwne dla $n = 4$.

nr	stan początkowy	długość ścieżki	IDA* odwiedzonych	IDA* czas [s]	A* stanów odwiedzonych i oczekujących	A* czas [s]
85	4,7,13,10,1,2,9,6,12,8,14,5,3,0,11,15	44	$1.5 \cdot 10^7$	12.3	$1.7 \cdot 10^5$, $1.6 \cdot 10^5$	0.9
5	4,7,14,13,10,3,9,12,11,5,6,15,1,2,8,0	56	$2.6 \cdot 10^7$	20.4	$1.6 \cdot 10^6$, $1.4 \cdot 10^6$	11.7
2	13,5,4,10,9,12,8,14,2,3,7,1,0,15,11,6	55	$3.8 \cdot 10^7$	31.2	$2.6 \cdot 10^6$, $2.1 \cdot 10^6$	26.9
54	12,11,0,8,10,2,13,15,5,4,7,3,6,9,14,1	56	$1.9 \cdot 10^8$	150.5	brak RAM (2 GB) przy: $3.1 \cdot 10^6$, $2.5 \cdot 10^6$	—
1	14,13,15,7,11,12,9,5,6,0,2,1,4,8,10,3	57	$2.5 \cdot 10^8$	212.3	brak RAM (2 GB) przy: $3.4 \cdot 10^6$, $2.8 \cdot 10^6$	—

wykonania (nawet rzędu 2–3 minut). Algorytm A* pracował istotnie krócej, przy czym w dwóch ostatnich przypadkach doprowadził do nieprawidłowego zakończenia, wyczerpując całą dostępną pamięć RAM.

Algorytm 7 IDA*

```
1:  procedura IterativeDeepeningAStar(s_0)                    ▷ stan początkowy: s_0
2:      g(s_0) := 0                                           ▷ droga przebyta od startu
3:      oblicz h(s_0)                                         ▷ heurystyka wg podanego przepisu
4:      f(s_0) := g(s_0) + h(s_0)
5:      ustaw pusty wskaźnik na rodzica dla s_0
6:      Open := {s_0}                                         ▷ kolejka stanów oczekujących
7:      H := f(s_0), H' := ∞                                  ▷ początkowy i następny horyzont
8:      dopóki Open ≠ ∅ wykonaj
9:          pobierz (i usuń) z Open stan s o najmniejszej wartości f(s)   ▷ operacja „poll"
10:         jeżeli g(s) > H to
11:             H' := min{H', f(s)}
12:             jeżeli Open = ∅ to
13:                 H := H', H' := ∞, Open := {s_0}           ▷ pogłębianie horyzontu
14:             kontynuuj od następnej iteracji
15:         jeżeli s jest stanem końcowym to zwróć s          ▷ znaleziono rozwiązanie
16:         wygeneruj zbiór stanów {t} potomnych dla s
17:         dla wszystkich t wykonaj
18:             g(t) := g(s) + Δ(s → t)
19:             oblicz h(t)
20:             f(t) := g(t) + h(t)
21:             ustaw wskaźnik rodzica t na s
22:             jeżeli t ∉ Open to
23:                 dodaj t do Open
24:             w przeciwnym razie
25:                 jeżeli nowy koszt g(t) jest mniejszy niż znany dotychczas to
26:                     zastąp t w Open nowym egzemplarzem (aktualnie badanym)
27:                     uaktualnij pozycję t w Open
28:     zwróć null                                           ▷ nie znaleziono rozwiązania
```

2.4 Ćwiczenia laboratoryjne (Java + biblioteka *SaC*)

Ćwiczenie 2.1 **Napisz program rozwiązujący łamigłówkę sudoku z wykorzystaniem algorytmu best-first search i heurystyki „liczba niewiadomych".** Wskazówki: napisz klasę `Sudoku` reprezentującą stan planszy sudoku (klasa powinna być ogólna, tzn. powinna pozwalać na reprezentację plansz sudoku dowolnych rozmiarów $n^2 \times n^2$, domyślnie $n = 3$) — wybierz odpowiedni typ tablicowy; przygotuj konstruktor domyślny i konstruktor kopiujący; przygotuj metody pozwalające na: zwrócenie napisowej reprezentacji stanu — `toString()`, wczytanie sudoku w wersji 9×9 z napisu, sprawdzenie poprawności planszy, zliczenie liczby niewiadomych; wykonaj dziedziczenie z klasy `sac.graph.GraphStateImpl`; dostarcz implementacje metod `isSolution()` oraz `generateChildren(...)` (generując stany potomne można wybrać dowolną komórkę tablicy z niewiadomą); zaimplementuj klasę reprezentującą heurystykę „liczba niewiadomych" (klasa heurystyki powinna dziedziczyć po klasie `sac.StateFunction` i zawierać implementację metody `calculate(...)`); podepnij heurystykę do klasy `Sudoku` za pomocą metody statycznej `setHFunction(...)`; w celu identyfikacji stanów w zbiorze *Closed* dostarcz implementację metody `hashCode()`, zwracając kod mieszający na podstawie napisowej reprezentacji planszy (zapoznaj się ze standardowym działaniem metody `hashCode()` na rzecz obiektów klasy `java.lang.String`); napisz właściwy program w metodzie `main(...)`, a w nim stwórz początkowy stan sudoku zasilony z napisu i uruchom rozwiązywanie sudoku z wykorzystaniem algorytmu reprezentowanego przez klasę `sac.graph.BestFirstSearch`; sprawdź prawidłowość działania dla kilku przykładów; poza rozwiązaniem wypisz dodatkowo na ekran inforamcje na temat: czasu rozwiązywania, liczby stanów w zbiorach *Open* i *Closed* w chwili stopu. Przykładowe plansze sudoku 9×9 w formie tekstowej można znaleźć np. pod adresem: `https://projecteuler.net/project/resources/p096_sudoku.txt`.

Ćwiczenie 2.2 **Modyfikując odpowiednio początkową planszę sudoku znajdź więcej niż jedno rozwiązanie (wykorzystaj program z Ćwiczenia 2.1).** Wskazówki: wykorzystaj klasę `sac.graph.GraphSearchConfigurator`, zmieniając tak warunek stopu, aby algorytm zatrzymywał się dopiero po napotkaniu 2 rozwiązań; wykryj drugie rozwiązanie, odbierając stopniowo wiadome z początkowej planszy sudoku i uruchamiając program rozwiązujący (uwaga: obserwuj po każdym uruchomieniu liczby stanów w zbiorach *Open* i *Closed*); ponownie zmień nastawy konfiguracyjne, żądając aby algorytm zatrzymał się po napotkaniu maksymalnej możliwej liczby rozwiązań (stała: `Integer.MAX_VALUE`).

Ćwiczenie 2.3 **Wypisz na ekran wszystkie rozwiązania sudoku dla planszy 4×4 (wykorzystaj program z Ćwiczenia 2.1).** Wskazówka: zmniejsz odpowiednio wymiarowość plansz i wykorzystaj doświadczenia z Ćwiczenia 2.2.

Ćwiczenie 2.4 **Przyspiesz program rozwiązujący sudoku poprzez zwiększenie wydajności reprezentacji napisowych i generowania kodów mieszających.** Wskazówki: w metodzie `toString()` wykorzystaj klasę `java.lang.StringBuilder` do budowania napisów zamiast standardowej klasy `java.lang.String` i operatora `+`; rozważ możliwość generowania kodów mieszających bezpośrednio na podstawie zawartości tablicy z planszą — patrz `java.util.Arrays.hashCode(...)` (uwaga: zwróć uwagę na wymiarowość tablic); zmierz uzyskane przyspieszenia (zadając przeszukiwanie wymagające czasowo) i sprawdź prawidłowość wyników.

Ćwiczenie 2.5 **Ulepsz program rozwiązujący sudoku poprzez generowanie potomków w „komórce minimalnej”.** Wskazówki: uzupełnij klasę `Sudoku` o odpowiednią strukturę, która pozwoli śledzić pozostałe możliwości (cyfry) dla każdej komórki planszy; generując stany potomne w metodzie `generateChildren()` wybieraj jedną z komórek o najmniejszej liczbie pozostałych możliwości; pamiętaj o kopiowaniu informacji o pozostałych możliwościach w konstruktorze kopiującym; porównaj działanie nowej i starej wersji programu rozwiązującego (czasy wykonania, liczba odwiedzanych stanów).

Ćwiczenie 2.6 **Zaimplementuj dodatkową heurystykę „suma pozostałych możliwości” do programu rozwiązującego sudoku.** Wskazówki: wykorzystując dodatkowe informacje wprowadzone do klasy `Sudoku` w Ćwiczeniu 2.5 zaimplementuj dodatkową funkcję heurystyczną „suma pozostałych możliwości”; porównaj działanie obu heurystyk na kilku przykładach (czasy wykonania, liczba odwiedzanych stanów); przygotuj większy eksperyment statystyczny w ramach metody `main(...)`, który porówna dwie heurystyki dla przynajmniej 100 przykładów (w pętli podpinaj naprzemiennie heurystyki metodą `setHFunction(...)` dla każdej planszy początkowej).

Ćwiczenie 2.7 **Napisz program rozwiązujący układankę „puzzle przesuwne” z wykorzystaniem algorytmu A* oraz heurystyk „kafelki na niewłaściwym miejscu” i „Manhattan”.** Wskazówki: napisz klasę `SlidingPuzzle` reprezentującą stan planszy układanki „puzzle przesuwne”, postępując zgodnie z ogólnymi wytycznymi z Ćwiczenia 2.1 (parametryzowana wymiarowość, konstruktory, `toString()`, `hashCode()`, dziedziczenie z klasy `GraphStateImpl` itd.); przygotuj metodę wykonującą pojedynczy ruch oraz metodę generującą pomieszaną planszę (do wielokrotnego wykonywania losowych ruchów wykorzystaj obiekt klasy `java.util.Random`); generując stany potomne w metodzie `generateChildren()` użyj na rzecz każdego potomka metody `setMoveName(...)`, pozwalającej nadać mu nazwę wg kierunku wykonanego ruchu, np. `L`, `R`, `U`, `D` (będzie to przydatne dalej przy wypisie ścieżki ruchów); przygotuj dwie klasy reprezentujące heurystyki „kafelki na niewłaściwym miejscu” i „Manhattan”; przygotuj dwa warianty metody `main(...)` — wariant pierwszy pozwalający rozwiązać pojedynczą układankę za pomocą algorytmu A* (klasa `sac.graph.AStar`) i wypisać dla niej ścieżkę ruchów, oraz wariant drugi wykonujący statystyczne porównanie dwóch heurystyk; w ramach drugiego wariantu wygeneruj

100 losowych plansz początkowych (każda pomieszana za pomocą 1000 ruchów) i każdą z nich rozwiąż dwukrotnie przełączając się pomiędzy heurystykami — `setHFunction(...)`, oblicz i wyświetl średnią liczbę stanów odwiedzanych przez każdą z heurystyk oraz średnie czasy wykonań.

E **Ćwiczenie 2.8 Porównaj działanie algorytmów A* i best-first search rozwiązujących „puzzle przesuwne".** Wskazówki: wykonaj eksperyment statystyczny (analogicznie jak w Ćwiczeniu 2.7) rozwiązując każdą z plansz początkowych dwukrotnie algorytmami A* i best-first search (przy ustalonej heurystyce); podmianę klasy z algorytmem można „zautomatyzować" poprzez użycie referencji na obiekt ogólnej klasy o nazwie `sac.graph.GraphSearchAlgorithm` i odpowiednie podstawianie do niego obiektów klas `sac.graph.AStar` lub `sac.graph.BestFirstSearch` przebywających np. w dwuelementowej tablicy (pętla po algorytmach); w ramach porównania obserwuj: średni czas, średnią liczbę odwiedzanych stanów, i średnią długość znalezionej ścieżki.

E **Ćwiczenie 2.9 Zaimplementuj trzecią heurystykę „Manhattan + konflikty liniowe" do programu rozwiązującego „puzzle przesuwne.** Wskazówki: zaimplementuj dodatkową trzecią funkcję heurystyczną „Manhattan + konflikty liniowe", która doliczy do podstawowego składnika Manhattan dwa ruchy za każdy obecny na planszy konflikt liniowy (uwaga: zgodnie z informacjami podanymi w sekcji 2.3.2, zlicz konflikty liniowe w wierszach i kolumnach bez nadmiarowości); porównaj działanie nowej heurystyki z poprzednimi poprzez odpowiedni eksperyment statystyczny (analogicznie jak w Ćwiczeniu 2.7).

2.5 Ćwiczenia laboratoryjne (C# + biblioteka *AISearch*)

E **Ćwiczenie 2.10 Napisz program rozwiązujący łamigłówkę sudoku z wykorzystaniem algorytmu best-first search i heurystyki „liczba niewiadomych".** W trakcie implementacji należy utworzyć dwie klasy potomne `SudokuState.cs` i `SudokuSearch.cs` dziedziczące odpowiednio po klasach bazowych `State.cs` i `BestFirstSearch.cs`. Klasa `SudokuState` będzie reprezentować pojedynczy stan planszy sudoku. Zadaniem klasy `SudokuSearch` będzie zastosowanie algorytmu best-first search do rozwiązania konkretnej planszy Sudoku. Dobra praktyka programowania mówi, że pojedynczy plik powinien zawierać w sobie implementację pojedynczej klasy. Szczegółowe wskazówki implementacyjne zamieszczonowo w dodatku 11.2. Należy przetestować działanie algorytmu best-first search dla kilku przykładowych sudoku. Poza rozwiązaniem należy wyświetlić różne informacje na temat pracy algorytmu w momencie zatrzymania: liczba stanów w zbiorach *Open* i *Closed*, czas pracy.

Ćwiczenie 2.11 **Modyfikując odpowiednio początkową planszę sudoku znajdź więcej niż jedno rozwiązanie (wykorzystaj program z Ćwiczenia 2.10).** Wskazówki: zmodyfikuj konstruktor klasy `SudokuSearch` aby konstruktor klasy bazowej otrzymywał informację o liczbie rozwiązań do odnalezienia:

```
public SudokuSearch(SudokuState state, int
    numberOfSolutions) : base(state, numberOfSolutions) { }
```

Wykryj drugie rozwiązanie, odbierając stopniowo wiadome z początkowej planszy sudoku i uruchamiając program rozwiązujący (uwaga: obserwuj po każdym uruchomieniu liczby stanów w zbiorach *Open* i *Closed*). Przetestuj program żądając aby algorytm zatrzymał się po napotkaniu maksymalnej możliwej liczby rozwiązań (stała: `int.MaxValue`).

Ćwiczenie 2.12 **Wypisz na ekran wszystkie rozwiązania sudoku dla planszy 4×4 (wykorzystaj program z Ćwiczenia 2.10).** Wskazówka: zmniejsz odpowiednio wymiarowość plansz i wykorzystaj doświadczenia z Ćwiczenia 2.11.

Ćwiczenie 2.13 **Ulepsz program rozwiązujący sudoku poprzez generowanie potomków w „komórce minimalnej".** Wskazówki: uzpełnij klasę potomną dziedziczącą po klasie `State` o odpowiednią strukturę, która pozwoli śledzić pozostałe możliwości (cyfry) dla każdej komórki planszy. Generując stany potomne w metodzie `buildChildren()` wybieraj jedną z komórek o najmniejszej liczbie pozostałych możliwości. Pamiętaj o kopiowaniu informacji o pozostałych możliwościach do stanów potomnych. Porównaj działanie nowej i starej wersji programu rozwiązującego (czasy wykonania, liczba odwiedzanych stanów).

Ćwiczenie 2.14 **Zaimplementuj dodatkową heurystykę „suma pozostałych możliwości" do programu rozwiązującego sudoku.** Wskazówki: wykorzystując dodatkowe informacje wprowadzone do klasy `Sudoku` w Ćwiczeniu 2.13 zaimplementuj dodatkową funkcję heurystyczną „suma pozostałych możliwości". Porównaj działanie obu heurystyk na kilku przykładach (czasy wykonania, liczba odwiedzanych stanów). Przygotuj większy eksperyment statystyczny w ramach metody `Main`, który porówna dwie heurystyki dla przynajmniej 100 przykładów.

Ćwiczenie 2.15 **Napisz program rozwiązujący układankę „puzzle przesuwne" z wykorzystaniem algorytmu A* oraz heurystyk „kafelki na niewłaściwym miejscu" i „Manhattan".** Wskazówki: należy utworzyć dwie klasy potomne `PuzzleState.cs` i `PuzzleSearch.cs` dziedziczące po klasach bazowych `State.cs` i `AStarSearch.cs`. Implementacja jest analogiczna jak w ćwiczeniu 2.10. Zasadnicza różnica, o której należy pamiętać, dotyczy konstruktorów klasy `PuzzleState`, w której należy zdefiniować już przebytą drogę do danego stanu. Informacja ta jest wymagana do poprawnego działania algorytmu A*.

```
public PuzzelState(PuzzelState parent, ... /*pozostale
    parametry*/) : base(parent) {
    //cialo konstruktora

    this.h = ComputeHeuristicGrade();
    //W stanie potomnym droga ktora przebylismy jest o
        jeden wieksza niz w rodzicu
    this.g = parent.g + 1;
}
```

Jako potomków stanu reprezentującego puzzle o układzie:

2	1	6
	5	7
3	8	4

rozumiemy następujące stany:

	1	6
2	5	7
3	8	4

2	1	6
5		7
3	8	4

2	1	6
3	5	7
	8	4

Należy zaimplementować dwie funkcje heurystyczne: „kafelki na niewłaściwym miejscu” oraz „Manhattan”. Należy przygotować dwie wersje metody `Main`:

- Wariant pierwszy pozwalający rozwiązać pojedynczą układankę za pomocą algorytmu A* i wypisać dla niej poszczególne stany prowadzące do rozwiązania.
- Wariant drugi wykonujący statystyczne porównanie dwóch heurystyk. W ramach drugiego wariantu wygeneruj 100 losowych plansz początkowych (każda pomieszana za pomocą 1000 ruchów) i każdą z nich rozwiąż dwukrotnie przełączając się pomiędzy heurystykami Oblicz i wyświetl średnią liczbę stanów.

Należy pamiętać, że generowanie zupełnie losowego układu planszy może powodować powstanie planszy, której nie da się rozwiązać. Przykładem takiego układu w planszy 2x2 jest:

1	3
2	

Powyższy układ planszy nie da się doprowadzić do postaci:

1	2
3	

Podobnego typu układy występują w większych planszach, toteż zaleca się, aby konstruktor tworzący puzzle przyjmował jedynie dwa parametry: rozmiar planszy i liczbę mieszań. Wewnątrz konstruktora powinna zostać utworzona tablica z ułożonymi puzzlami,

która następnie zostanie pomieszana. „Losowe” układanki należy generować, rozpoczynając od ułożonej planszy i wykonując zadaną w konstruktorze *liczbę mieszań*, nie dbając o ewentualne niwelowanie się ruchów przeciwnych, natomiast dbając o nie zliczanie się ruchów pustych przy brzegach planszy.

E **Ćwiczenie 2.16** **Porównaj działanie algorytmów A* i best-first search rozwiązujących „puzzle przesuwne”.** Wskazówki: wykonaj eksperyment statystyczny (analogicznie jak w Ćwiczeniu 2.15) rozwiązując każdą z plansz początkowych dwukrotnie algorytmami A* i best-first search (przy ustalonej heurystyce). W ramach eksperymentu obserwuj: średni czas, średnią liczbę odwiedzanych stanów i średnią długość znalezionej ścieżki.

E **Ćwiczenie 2.17** **Zaimplementuj trzecią heurystykę „Manhattan + konflikty liniowe” do programu rozwiązującego „puzzle przesuwne”.** Wskazówki: zaimplementuj dodatkową trzecią funkcję heurystyczną „Manhattan + konflikty liniowe”, która doliczy do podstawowego składnika Manhattan dwa ruchy za każdy obecny na planszy konflikt liniowy (uwaga: zgodnie z informacjami podanymi w sekcji 2.3.2, zlicz konflikty liniowe w wierszach i kolumnach bez nadmiarowości). Porównaj działanie nowej heurystyki z poprzednimi poprzez odpowiedni eksperyment statystyczny (analogicznie jak w Ćwiczeniu 2.7).

2.6 Ćwiczenia laboratoryjne (C++ + biblioteka *SI++*)

E **Ćwiczenie 2.18** **Napisz program rozwiązujący łamigłówkę sudoku z wykorzystaniem algorytmu best-first search i heurystyki „liczba niewiadomych”.** Wskazówki: napisz klasę szablonową `generic_sudoku` reprezentującą stan planszy sudoku o rozmiarze $mn \times mn$ (wymiary przekaż w parametrach szablonu, domyślnie $m = 3$ i $n = 3$) — wybierz odpowiedni typ tablicowy (`std::array`); wykonaj dziedziczenie z klasy `graph_state`; przygotuj konstruktor przyjmujący tablicę reprezentującą planszę; dostarcz implementacje metod: `clone()`, `hash_code()`, `get_successors()` (generując stany potomne wybierz dowolną pustą komórkę; wolno generować wyłącznie poprawne stany), `is_solution()`, `to_string()` oraz `is_equal()`; w miarę potrzeb można dodać własne metody; mając gotową klasę stwórz nową klasę szablonową dziedziczącą po niej i podmień implementacje metod `clone()` oraz `get_heuristic_grade()`, która zwróci informację o liczbie niewiadomych; napisz właściwy program w funkcji `main()`, a w nim stwórz początkowy stan sudoku zasilony z tablicy (rozważ napisanie konstruktora, który przyjmie napis reprezentujący diagram sudoku) i uruchom rozwiązywanie sudoku konstruując obiekt klasy `informative_searcher`, któremu w konstruktorze oprócz stanu początkowego przekażesz komparator; sprawdź poprawność działania dla kilku przykładów; poza rozwiązaniem wypisz dodatkowo na ekran informacje na temat: czasu rozwiązywania, liczby stanów w zbiorach *Open* i *Closed* w chwili stopu; same klasy mogą wyglądać następująco:

```
template<int M, int N>
class generic_sudoku : public graph_state
{
// ...
};

template<int M, int N, typename Heuristic>
class sudoku_state : public generic_sudoku<M, N>
{
// ...
private:
    static constexpr Heuristic heuristic {};
};

template<int M, int N>
struct H_remaining
{
    double operator()(/* ... */) const
    {
        return 0;
    }
};
```

komparator można zdefiniować następująco:

```
auto comp = [](const graph_state &a, const graph_state &b)
{
    return a.get_h() < b.get_h();
};
```

zaś metoda `is_equal()` może wyglądać tak (przyjmując, że `board` to obiekt klasy `std::array`):

```
bool is_equal(const graph_state &s) const override
{
    const generic_sudoku *st = dynamic_cast<const
        generic_sudoku*>(&s);
    return st != nullptr && st->board == this->board;
}
```

Ćwiczenie 2.19 **Modyfikując odpowiednio początkową planszę sudoku znajdź więcej niż jedno rozwiązanie (wykorzystaj program z Ćwiczenia 2.18).** Wskazówka: konstruktor klasy `informative_searcher` przyjmuje trzeci parametr określający liczbę rozwiązań do znalezienia — przekaż wartość `std::numeric_limits<size_t>::max()`.

Ćwiczenie 2.20 **Wyznacz liczbę wszystkich rozwiązań sudoku dla planszy 6×6 — $M = 2$, $N = 3$ (wykorzystaj program z Ćwiczenia 2.18).** Wskazówka: dokonaj obliczeń w sposób pośredni, tzn. wyznacz liczbę rozwiązań dla planszy, której pierwszy wiersz zawiera cyfry `1,2,3,4,5,6` (wykorzystaj doświadczenia z Ćwiczenia 2.19), a uzyskaną liczbę rozwiązań przemnóż przez wartość `6!` (liczba permutacji cyfr wiersza).

Ćwiczenie 2.21 **Ulepsz program rozwiązujący sudoku poprzez generowanie potomków w „komórce minimalnej”.** Wskazówki: stwórz nową klasę dziedziczącą po `sudoku_state` i podmień implementacje metod `clone()` i `get_successors()`, w której znajdziesz komórkę z najmniejszą liczbą możliwości wypełnienia; porównaj działanie nowej i starej wersji programu rozwiązującego (czasy wykonania, liczba odwiedzanych stanów).

Ćwiczenie 2.22 **Zaimplementuj dodatkową heurystykę „suma pozostałych możliwości” do programu rozwiązującego sudoku.** Wskazówki: stwórz klasę podobną do `H_remaining`; porównaj działanie obu heurystyk dla kilku przykładów (czasy wykonania, liczba odwiedzanych stanów); przygotuj większy eksperyment statystyczny w ramach funkcji `main()`, który porówna dwie heurystyki dla przynajmniej 100 przykładów.

Ćwiczenie 2.23 **Napisz program rozwiązujący układankę „puzzle przesuwne” z wykorzystaniem algorytmu A* oraz heurystyk „kafelki na niewłaściwym miejscu” i „Manhattan”.** Wskazówki: stwórz generyczną klasę `sliding_puzzle` reprezentującą stan planszy układanki „puzzle przesuwne”, postępując zgodnie z ogólnymi wytycznymi z Ćwiczenia 2.18 (parametryzowana wymiarowość, konstruktory, dziedziczenie z klasy `graph_state` itd.); przygotuj metodę generującą pomieszaną planszę (do wielokrotnego wykonywania losowych ruchów użyj obiektów klasy `std::default_random_engine` oraz `std::uniform_int_distribution`); przygotuj klasy reprezentujące heurystyki „kafelki na niewłaściwym miejscu” i „Manhattan”; przygotuj dwie wersje funkcji `main()` — wariant pierwszy pozwalający rozwiązać pojedynczą układankę za pomocą algorytmu A* (obiekt klasy `informative_searcher` z odpowiednim komparatorem) i wypisać dla niej ścieżkę ruchów (przygotuj statyczną metodę, która przyjmie jako parametr wskaźnik na rozwiązanie, a w wyniku zwróci napis przedstawiający ruchy), oraz wariant drugi wykonujący statystyczne porównanie dwóch heurystyk; w ramach drugiego wariantu wygeneruj 100 losowych plansz początkowych (każda pomieszana za pomocą 1000 ruchów) i każdą z nich rozwiąż dwukrotnie, oblicz i wyświetl średnią liczbę stanów odwiedzanych przez każdą z heurystyk oraz średnie czasy wykonań; komparator można zdefiniować następująco:

```
auto comp = [](const graph_state &a, const graph_state &b)
{
    return a.get_f() < b.get_f();
};
```

E **Ćwiczenie 2.24** **Porównaj działanie algorytmów A* i best-first search rozwiązujących „puzzle przesuwne”.** Wskazówki: wykonaj eksperyment statystyczny (analogicznie jak w Ćwiczeniu 2.23) rozwiązując każdą z plansz początkowych dwukrotnie algorytmami A* i best-first search (przy ustalonej heurystyce); w ramach porównania obserwuj: średni czas, średnią liczbę odwiedzanych stanów i średnią długość znalezionej ścieżki.

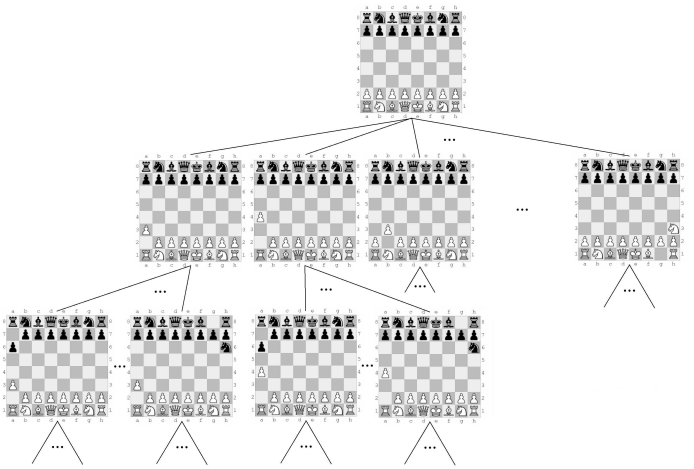
E **Ćwiczenie 2.25** **Zbadaj wpływ zmiany porządku odwiedzania stanów o równej wartości f.** Wskazówka: przygotuj komparator porównujący dwa stany `a` i `b` i zwracający prawdę, gdy $f_a < f_b \vee f_a = f_b \wedge h_a < h_b$; porównaj działanie nowego sposobu porządkowania z poprzednim poprzez odpowiedni eksperyment statystyczny (analogicznie jak w Ćwiczeniu 2.23).

3. Przeszukiwanie drzew gier

Algorytmy do przeszukiwania drzew gier są oparte na pojęciu *minimaksu* (ang. *minimax*). Historycznie pojęcie to można przypisać von Neumannanowi, który sformułował i udowodnił *twierdzenie o minimaksie* [Neu28; NM44]. Twierdzenie to samo w sobie ma trochę inny i bardziej ogólny kontekst niż ten, z którym spotykamy się w zagadnieniach gier (jak np. szachy). Mówiąc dokładniej, twierdzenie tyczy gier dwuosobowych o sumie zerowej, obejmuje przypadki, gdy gracze wykonują ruchy naprzemienne lub równoczesne, i implikuje istnienie tzw. *optymalnej strategii mieszanej* dla każdego z graczy. Jeżeli obaj gracze stosują swoje optymalne strategie, to gra zostanie doprowadzona do punktu minimaksowego (zwanego również punktem siodłowym), tj. punktu, w którym żaden z graczy nie może poprawić swojej wypłaty zmieniając strategię. Mówiąc jeszcze inaczej, termin minimaks można traktować także jako pewną regułę decyzyjną, która nakazuje graczowi minimalizować maksymalną możliwą wypłatę dla przeciwnika.

Zajmując się algorytmami do analizy drzew gier, zwykle rozpatrujemy pewne gry dwuosobowe umysłowe, takie jak np. szachy, warcaby, GO itp. Grę można zatem rozumieć jako pewną sytuację konfliktową, w której gracze mają sprzeczne interesy i gdzie mamy jasno zdefiniowane reguły. Z algorytmicznego punktu widzenia problem przeszukiwania drzewa gry można sformułować w sposób następujący: mając daną pewną pozycję w grze (w szczególności początkową), należy wysta-

wić ***oceny liczbowe*** dla poszczególnych ruchów (akcji) możliwych dla gracza, na którego przypada teraz kolej ruchu; ocena powinna reprezentować dokładną lub przybliżoną ***wypłatę*** (ang. *payoff*) gracza, jeżeli wybierze on dany ruch przy założeniu optymalnego postępowania drugiego gracza.



Rys. 3.1: Poglądowa ilustracja początkowego fragmentu drzewa gry dla szachów. Drzewo rośnie w tempie wykładniczym względem liczby poziomów, np. drugi poziom drzewa liczy już 400 stanów. (źródło: *opracowanie własne*)

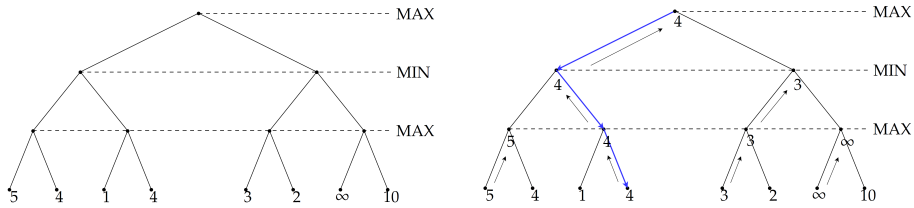
Warto w tym miejscu zwrócić od razu uwagę na dwa istotne elementy. Po pierwsze, w związku z wykładniczym wzrostem rozmiaru drzewa gry wraz z liczbą poziomów (patrz np. rys. 3.1), rzadko kiedy wypłaty obliczane przez algorytmy są dokładne. Najczęściej są to tylko wypłaty przybliżone (lub prawdopodobne) i zgodne z pewną ludzką wiedzą na temat danej gry. Funkcje obliczające takie oceny można także określać mianem heurystyk, przy czym sens tego słowa w grach będzie całkiem inny niż w przeszukiwaniach grafowych[1]. Obliczenie wypłat dokładnych jest w praktyce możliwe tylko dla prostych gier o małym drzewie gry (np. kółko i krzyżyk) lub też dla odpowiednio małych końcówek bardziej zaawansowanych gier (końcówki szachowe, warcabowe itp.). W takich przypadkach zbiór możliwych wypłat redukuje się często (dla wielu gier) do trzech możliwych wartości: wygrana, przegrana, remis. Drugi istotny element, to założenie o optymalnym postępowaniu drugiego gracza. Należy zaznaczyć, że założenie to nie powinno w ogólności psuć odpowiedzi algorytmów minimaksowych. Jeżeli drugi gracz nie postępuje w sposób optymalny, to gracz pierwszy, wykonując ruchy sugerowane przez algo-

[1]W szczególności, w grach heurystyczne funkcje oceny mogą zwracać wartości ujemne.

rytm, powinien tylko zyskiwać, tj. otrzymywać wypłaty większe lub równe tym, na które wskazuje algorytm (słowo „powinien” wynika z ograniczeń wcześniejszej uwagi i tzw. efektu horyzontu, który zostanie wyjaśniony później).

3.1 Algorytm min-max

Sposób działania algorytmu min-max (inne spotykane pisownie nazwy: minimax, minmax) można naszkicować następująco. Dla danej pozycji początkowej rozwijane jest drzewo gry do pewnej zadanej głębokości. Pozycjom końcowym (liściom, terminalom) nadawane są *oceny liczbowe*. Następuje „przechodzenie” drzewa od dołu propagując wybrane oceny w górę drzewa. W efekcie na końcu tego postępowania zostają ocenione możliwe ruchy pochodzące od stanu początkowego. Poglądowy schemat działania algorytmu przedstawiono na rys. 3.2.



Rys. 3.2: Schemat działania algorytmu min-max. (źródło: *opracowanie własne*)

Funkcja oceny pozycji (ang. *evaluation function*) jest zwykle pewną funkcją heurystyczną zgodną z wiedzą i intuicją ludzi na temat danej gry. Na przykład dla szachów, prosta funkcja oceny może obliczać różnicę pomiędzy materialną wartością bierek białych i czarnych (np. licząc piona jako 1 pkt., skoczki i gońce jako 3 pkt., wieże jako 5 pkt., i hetmana jako 9 pkt.). Bardziej zaawansowane funkcje powinny uwzględniać także elementy pozycyjne (np. kontrolę na centrum szachownicy, aktywność figur, bezpieczeństwo króla itp.).

W nomenklaturze związanej z algorytmami minimaksowymi, graczy biorących udział w grze nazywa się zwyczajowo *minimalizującym* i *maksymalizującym*. Zwycięstwo gracza minimalizującego reprezentuje $-\infty$, a zwycięstwo gracza maksymalizującego reprezentuje $+\infty$. W zapisach algorytmicznych wielkości te można traktować symbolicznie, ale równoważnie można o nich myśleć też programistycznie jako o pewnych skrajnych wartościach dostępnych w ramach danego typu liczbowego[2]. Wartość 0 reprezentuje zwyczajowo remis jako wynik pewnej

[2]Na przykład w języku Java: `Double.NEGATIVE_INFINITY`, `Double.POSITIVE_INFINITY` lub też $\pm$`Integer.MAX_VALUE`, odpowiednio dla liczb zmiennoprzecinkowych i całkowitych.

zakończonej gry lub też ocena stanu gry nie przejawiającego przewagi żadnego z graczy. Jak już wspomniano, gdy drzewo gry jest odpowiednio małe lub gdy badana jest ścisła końcówka pewnej gry i osiągnięte zostaną w drzewie faktyczne stany końcowe gry (zgodnie z jej regułami), to możliwe wartości liści wynoszą: $-\infty$, $+\infty$, 0. Wtedy heurystyczna ocena pozycji jest niepotrzebna (mamy oceny dokładne).

Przed zaprezentowaniem właściwego pseudokodu algorytmu min-max warto wyjaśnić jeszcze następujący zestaw pojęć i oznaczeń:

- **półruch** (ang. *ply* lub *half-move*) — nazwa oznaczająca ruch jednego z graczy; przesuwanie się o jeden poziom w drzewie liczone jest zwyczajowo jako $\pm\frac{1}{2}$, dopiero 2 półruchy każdego z graczy traktowane są jako całe posunięcie.
- **współczynnik rozgałęziania** (ang. *branching factor*) — przeciętna lub stała liczba ruchów przypadająca na każdego z graczy w danej grze; oznaczany zwykle literą b (np. dla szachów w grze środkowej $b \approx 40$).
- **horyzont przeszukiwań** — zadana do zbadania głębokość drzewa gry mierzona liczbą całych posunięć; oznaczany zwykle literą D (np. $D = 3.5$ odpowiada 7 półruchom i fizycznie 7 poziomom drzewa gry, nie licząc poziomu korzenia).
- **efekt horyzontu** — ogólna wada wszystkich procedur minimaksowych wynikająca z ograniczonej głębokości przeszukiwania; zjawisko polegające na tym, że pewien stan tuż poza horyzontem przeszukiwań może całkowicie zmieniać ocenę pozycji i np. okazać się katastrofalny dla gracza, pomimo że poziom wyżej pozycja była atrakcyjna (lub odwrotnie).
- **Quiescence** — technika pomocnicza łagodząca częściowo efekt horyzontu, polegająca na rozwijaniu stanów na granicy horyzontu przeszukiwań (i poza nim) aż do osiągnięcia tzw. *pozycji cichych* (np. nie zawierających możliwych zbić).

Algorytm 8 przedstawia pseudokod algorytmu min-max wyrażony w formie dwóch bliźniaczych procedur rekurencyjnych, które wywołują siebie nazwajem w sposób krzyżowy. Krótkiego wyjaśnienia wymaga występująca w algorytmie funkcja rutynowa IsTerminal(...). Jej zadaniem jest sprawdzenie, czy jesteśmy w pewnym punkcie stopu (tj. w stanie końcowym lub inaczej terminalnym), a jej implementacja wynika po części z reguł danej gry. Zwykle funkcja ta sprawdza, czy zachodzi którykolwiek z warunków:

- $d \geqslant D$ i stan s jest *cichy*,
- $h(s) = \pm\infty$ — stan s jest *zwycięski*,
- $h(s) \neq \pm\infty$, ale stan s jest *remisowym* wg zasad gry (np. w szachach: pat, wieczny szach, trzykrotne powtórzenie pozycji).

Algorytm 8 Min-max

```
1:  procedura MMEvaluateMaxState(s, d, D)
2:      jeżeli IsTerminal(s, d, D) to zwróć h(s)        ▷ h(s) — heurystyczna ocena pozycji
3:      v := −∞
4:      wygeneruj zbiór stanów {t} potomnych dla s
5:      dla wszystkich t wykonaj
6:          w :=MMEvaluateMinState(t, d + 1/2, D)
7:          jeżeli s jest korzeniem to zapamiętaj w jako ocenę ruchu s → t
8:          v := max{v, w}
9:      zwróć v
10: procedura MMEvaluateMinState(s, d, D)
11:     jeżeli IsTerminal(s, d, D) to zwróć h(s)        ▷ h(s) — heurystyczna ocena pozycji
12:     v := ∞
13:     wygeneruj zbiór stanów {t} potomnych dla s
14:     dla wszystkich t wykonaj
15:         w :=MMEvaluateMaxState(t, d + 1/2, D)
16:         jeżeli s jest korzeniem to zapamiętaj w jako ocenę ruchu s → t
17:         v := min{v, w}
18:     zwróć v
```

3.1.1 Funkcja oceny pozycji na przykładzie szachów

Jedną z pierwszych funkcji oceny pozycji szachowej była funkcja zaproponowana przez C. Shannona w 1949 r. Zawierała ona zarówno składniki materialne jak i pozycyjne, i miała następującą postać:

$$\begin{aligned} h(s) = {} & 200(K_s - K'_s) + 9(Q_s - Q'_s) + 5(R_s - R'_s) + 3(B_s - B'_s + N_s - N'_s) \\ & + 1(P_s - P'_s) - 0.5(D_s - D'_s + S_s - S'_s + I_s - I'_s) + 0.1(M_s - M'_s), \end{aligned} \tag{3.1}$$

gdzie K,Q,R,B,N,P oznaczają odpowiednio liczbę: króli, hetmanów, wieży, gońców, skoczków i pionów; D,S,I oznaczają liczbę pionów: podwojonych, zablokowanych, odizolowanych; M oznacza mobilność (liczbę dozwolonych ruchów). Symbole nieprimowane (bez znaczka $'$) oznaczają powyższe wielkości dla gracza maksymalizującego, zaś primowane dla gracza minimalizującego. Pewnego wyjaśnienia wymaga współczynnik wagowy 200 stojący przy królach. Prawdopodobnie w zamyśle Shannona był to odpowiednik „nieskończoności” w ramach przyjętej skali wartości. Innymi słowy, jeżeli któryś z graczy nie posiada króla w pewnej pozycji gry (pozycja znajdująca się dwa półruchy dalej niż pozycja matowa), to jest on „biedniejszy” o 200 punktów i ta duża różnica wskazuje na przegraną jego strony.

Współcześnie, szachowe funkcje oceny wyrażają swoje wartości w tzw. *centypionach*. Jeden pion ma wartość 100 centypionów, a pozostałe elementy są oceniane

relatywnie względem centypiona. W szczególności najmniejsza przewaga pozycyjna gracza to właśnie 1 centypion.

Elementów uwzględnianych w ocenie pozycji szachowej może być bardzo wiele, a do najpopularniejszych z nich należą:

- kontrola nad centrum,
- aktywność figur (i ich „łączność”[3]),
- struktura pionów,
- bezpieczeństwo króla,
- piony idące do przemiany,
- posiadana przestrzeń.

Siła gry pewnego programu, czyli tzw. sztucznej inteligencji, zależy bezpośrednio od jakości zaprojektowanej funkcji oceny. Jeżeli dla pewnej gry zaprojektujemy wadliwą funkcję, której oceny będą w pewnym stopniu lub w pewnych sytuacjach nieadekwatne do faktycznej natury gry, to sztuczna inteligencja będzie podejmowała błędne decyzje. Oczywistym przykładem może być np. przypisanie większej wartości wieży niż hetmanowi w szachach. Innym, mniej oczywistym, może być przypisanie warcabowej damce równowartości 10 pionów. Należy tu zauważyć, że program grający z powyższą „świadomością” będzie gotów poświęcić aż 9 pionów w celu zdobycia 1 damki, co może okazać się zgubne. Powyższe uwagi dotyczą funkcji oceny projektowanych ręcznie, określanych często angielskim terminem: *hand-crafted*. Odmiennymi od powyższego klasycznego podejścia są bardziej współczesne trendy polegające na próbach automatycznego wykrycia lub wyewoluowania właściwej funkcji oceny, m.in.: podejścia genetyczne, uczenie ze wzmocnieniem (ang. *reinforcement learning*), czy też uczenie głębokie (ang. *deep learning*). Z powyższych uwag wynika ogólne rozróżnienie na tzw. *słabą* i *silną* sztuczną inteligencję. Omawiane w niniejszym rozdziale klasyczne algorytmy i techniki należą do nurtu słabej sztucznej inteligencji, ponieważ w algorytmach tych zaszyta jest na stałe pewna ludzka wiedza na temat danego problemu (gry). Bardziej nowoczesne podejścia oczekują wypracowywania tzw. silnych sztucznych inteligencji, które bez jakiegokolowiek udziału człowieka, a tylko na podstawie rozegrania bardzo dużej liczby gier, „nauczą się” odpowiedniego wartościowania poszczególnych elementów gry[4].

3.1.2 Złożoność obliczeniowa algorytmu min-max

Zarówno sam zapis algorytmu min-max jak i dotychczasowe uwagi oraz ilustracje opisujące wykładniczy rozrost drzewa gry, pozwalają łatwo dostrzec, że w złożoności obliczeniowej algorytmu min-max w naturalny sposób pojawi się *suma ciągu*

[3] wzajemne wspieranie / ubezpieczanie się figur

[4] Można tu wspomnieć m.in. o projektach: *AlphaZero* i *AlphaGo*.

geometrycznego. Rolę ilorazu tego ciągu będzie pełnił współczynnik rozgałęziania gry — b. Liczba składników sumy będzie odpowiadała liczbie poziomów drzewa, przy czym dla notacyjnego uproszczenia przestaniemy na chwilę mierzyć głębokość połówkami i przyjmiemy całkowity indeks głębokości — $d = 0, 1, 2, \ldots$.

Pokazaną poniżej krótką analizę przeprowadzono podejściem rekurencyjnym. Dla algorytmu min-max nie jest to podejście konieczne i jedyne, ale wybieramy ten właśnie sposób, dlatego że przyda się ono później do bardziej skomplikowanej analizy złożoności algorytmu „przycinanie α-β", w którym niektóre fragmenty drzewa gry są odcinane (pomijane).

Niech R_d oznacza liczbę stanów, które trzeba odwiedzić w drzewie o głębokości d, aby poznać dokładną wartość danego stanu gry — stanu, który ukorzenia to drzewo. Ze względu na fakt, że algorytm min-max przegląda drzewo w sposób wyczerpujący (nie odcina żadnych poddrzew), wielkość R_d jest określona rekurencją:

$$\begin{aligned} R_0 &= 1; \\ R_d &= 1 + bR_{d-1}, \quad \text{dla } d > 0; \end{aligned} \tag{3.2}$$

którą można rozwinąć w następujący sposób

$$\begin{aligned} R_d &= 1 + bR_{d-1} \\ &= 1 + b(1 + bR_{d-2}) = 1 + b + b^2 R_{d-2} \\ &\vdots \\ &= 1 + b + b^2 + \cdots + b^d R_{d-d} = \frac{b^{d+1} - 1}{b - 1} \\ &< \frac{b^{d+1}}{b-1} = \underbrace{\frac{b}{b-1}}_{\leqslant 2} b^d \leqslant 2b^d \sim O(b^d) \end{aligned} \tag{3.3}$$

Mówiąc w uproszczeniu, pojawia się tu schemat: $\underbrace{O(b \cdot b \cdots b)}_{d}$

czyli d-krotne mnożenie współczynnika b.

3.2 „Przycinanie α-β"

Kilka osób jest uważanych za niezależnych i równoczesnych (niemalże) odkrywców algorytmu o nazwie „przycinanie α-β" (ang. *alpha-beta pruning* lub *alpha-beta cut-offs*). Odkrycia te miały miejsce na przełomie lat 50.–60. dwudziestego stulecia. W szczególności odkrywcami tymi byli: Daniel J. Edwards, Allen Newell, Hebert A. Simon, John McCarthy, Arthur Samuel, Alexander Brudno; patrz m.in. [Bru63;

EH63; NS76]. Później, w roku 1975, Knuth i Moore oczyścili nieco algorytm i podali szczegółową analizę jego złożoności obliczeniowej w artykule [KM75]. Pearl dowiódł optymalności tego algorytmu w roku 1982 [Pea82].

Algorytm „przycinanie α-β" zalicza się do ogólnej klasy metod *podziału i ograniczeń* (ang. *branch and bound*). Obejmuje ona metody, które poszukują rozwiązania pewnego problemu, generując różne możliwości jako rozgałęzienia w drzewie (*branch*) oraz formułując odpowiednie ograniczenia nierównościowe (*bound*), które pozwalają odrzucać niektóre fragmenty tego drzewa — takie fragmenty, które nie są w stanie poprawić najelpszego wykrytego dotychczas wyniku.

W trakcie analizy drzewa śledzone będą dwie wielkości α i β, których sens liczbowy jest następujący:

- α — gwarantowana dotychczas[5] wypłata gracza maksymalizującego,
- β — gwarantowana dotychczas wypłata gracza minimalizującego.

Przy najbardziej zewnętrznym wywołaniu rekurencyjnym dla korzenia drzewa zadaje się $\alpha = -\infty$, $\beta = \infty$, czyli najbardziej pesymistyczne „wartości" odpowiednie dla każdego z graczy.

Pseudokod „przycinania α-β" prezentuje Algorytm 9. Stany potomne (i ich

Algorytm 9 Przycinanie α-β

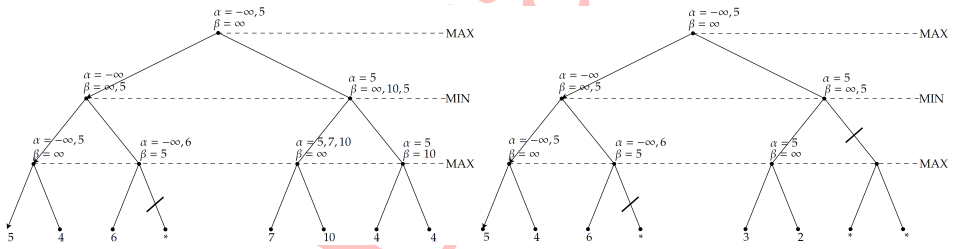
```
1:  procedura AlphaBetaEvaluateMaxState(s, d, D, α, β)
2:      jeżeli IsTerminal(s, d, D) to zwróć h(s)        ▷ h(s) — heurystyczna ocena pozycji
3:      wygeneruj zbiór stanów {t} potomnych dla s
4:      dla wszystkich t wykonaj
5:          v := AlphaBetaEvaluateMinState(t, d + 1/2, D, α, β)
6:          jeżeli s jest korzeniem to zapamiętaj v jako ocenę ruchu s → t
7:          α := max{α, v}
8:          jeżeli α ⩾ β to zwróć α                     ▷ przycięcie (!) — kolejne t nie będą sprawdzane
9:      zwróć α
10: procedura AlphaBetaEvaluateMinState(s, d, D, α, β)
11:     jeżeli IsTerminal(s, d, D) to zwróć h(s)        ▷ h(s) — heurystyczna ocena pozycji
12:     wygeneruj zbiór stanów {t} potomnych dla s
13:     dla wszystkich t wykonaj
14:         v := AlphaBetaEvaluateMaxState(t, d + 1/2, D, α, β)
15:         jeżeli s jest korzeniem to zapamiętaj v jako ocenę ruchu s → t
16:         β := min{β, v}
17:         jeżeli α ⩾ β to zwróć β                     ▷ przycięcie (!) — kolejne t nie będą sprawdzane
18:     zwróć β
```

poddrzewa) są podczas algorytmu analizowane dopóki spełniony pozostaje warunek $\alpha < \beta$. W momencie gdy zachodzi $\alpha \geqslant \beta$, algorytm przestaje rozpatrywać

[5] po wykonaniu dotychczasowej analizy i osiągnięciu pewnego punktu drzewa

kolejnych potomków danego stanu (i ich poddrzewa), ponieważ nie będą one miały wpływu na końcowy wynik. Innymi słowy takie przypadki byłyby wynikiem nieoptymalnego postępowania któregoś z graczy. Należy zauważyć, że przypadek nierówności ostrej $\alpha > \beta$ jest logiczną sprzecznością, ponieważ w żadnym momencie gry nie może być prawdą, że gracz maksymalizujący ma zagwarantowaną wypłatę większą niż gracz minimalizujący. Przypadek równości $\alpha = \beta$ nie stanowi co prawda sprzeczności, ale dla podniesienia wydajności algorytmu można go również dołączyć do wykluczeń, ponieważ nie wniesie on poprawy dotychczasowego wyniku.

Rysunek 3.3 ilustruje dwa przykłady działania „przycinania α-β”. Dla każdego stanu na rysunku rekurencyjne przeglądanie jego potomków odbywa się od lewej do prawej. Przy każdym ze stanów podano chronologicznie kolejne wartości przypisywane do zmiennych α i β. Ukośne przekreślenia na gałęziach wskazują pominięte poddrzewa. Gwiazdki oznaczają dowolne wartości, które nie mają wpływu na wartość gry w stanie początkowym (w korzeniu) i tym samym na wybór najlepszego ruchu. Zachęcamy czytelnika do samodzielnego prześledzenia tych przykładów oraz próby uzasadnienia, dlaczego pojawiły się poszczególne odcięcia.



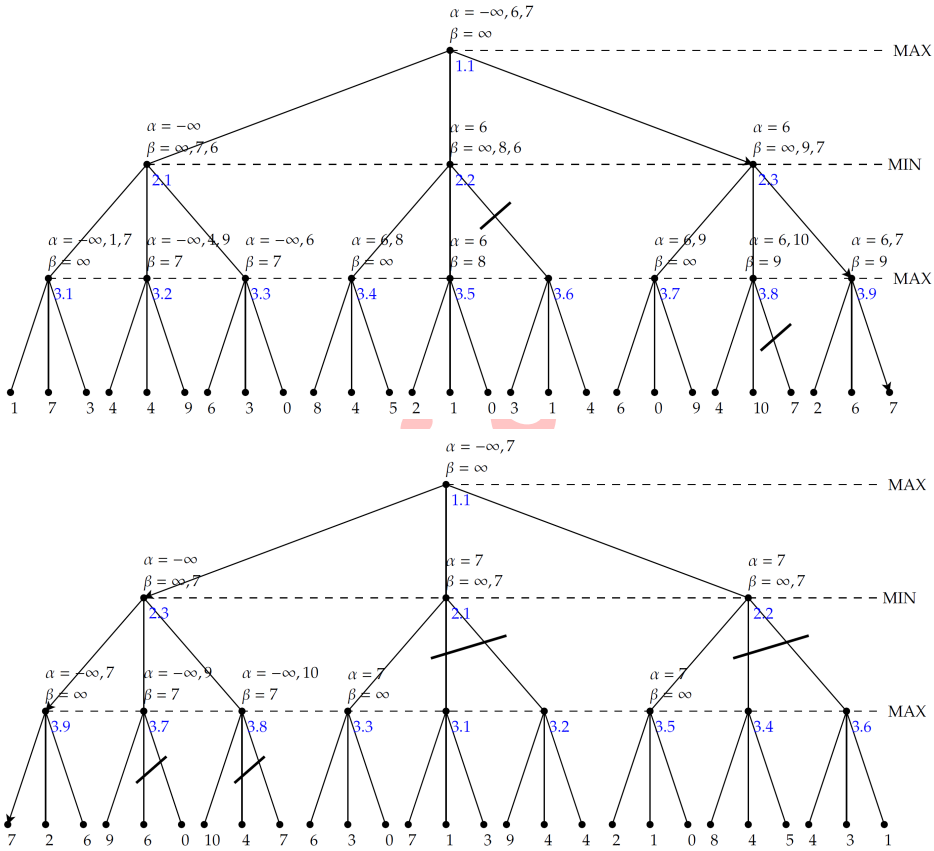
Rys. 3.3: Przykłady działania algorytmu „przycinanie α-β”. (źródło: *opracowanie własne*)

Pomimo redukcji drzewa algorytm „przycinanie α-β” oddaje te same wyniki (oceny ruchów) co algorytm min-max.

3.2.1 Złożoność obliczeniowa „przycinania α-β”

Złożoność obliczeniowa „przycinania α-β” jest zależna od *porządku* odwiedzania stanów potomnych. Sprzyjają sytuacje, gdy potomek powodujący odcięcie jest bliżej początku listy. Istnieją pewne techniki pomocnicze (np. odpowiednio sortujące potomków), które starają się zwiększyć częstość przycieć. Niemniej, w ogólności dobry porządek stanów potomnych nie jest znany z góry.

Ilustrację tego zagadnienia stanowi rys. 3.4. Porządek potomków wystepujący w górnym wariancie podanego drzewa gry prowadzi do dwóch odcięć i redukcji tylko 5 stanów całego drzewa. W wariancie drzewa pokazanym w dolnej części rysunku dla każdej listy potomków na pierwszym miejscu umieszczono stan o najlepszej wypłacie (z punktu widzenia danego gracza). Prowadzi to do czterech odcięć oraz redukcji aż 20 stanów całego drzewa.



Rys. 3.4: „Przycinanie α-β" — przykład różnych redukcji drzewa w zależności od porządku potomków. (źródło: *opracowanie własne*)

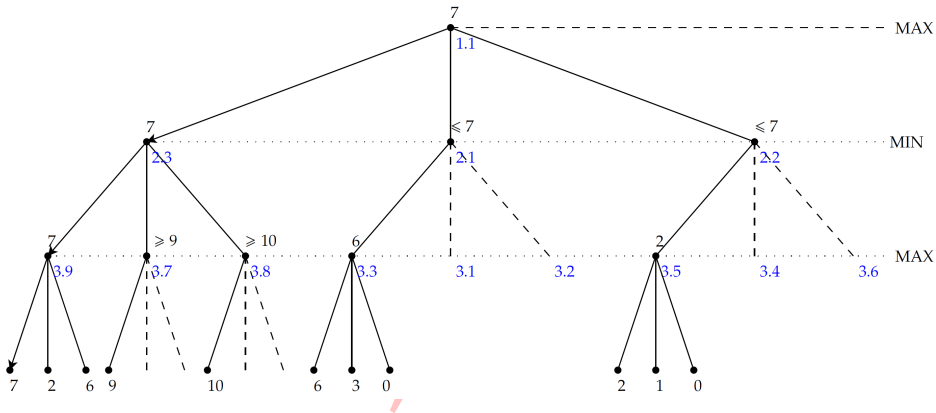
Można zatem uświadomić sobie dwa skrajne przypadki:

– „przycinanie α-β" może nie wykonać żadnego odcięcia i tym samym odwiedzi tyle samo stanów, co algorytm min-max,
– „przycinanie α-β" wykona największą możliwą liczbę odcięć (zgodnie ze wzorcem obserwowalnym w dolnej części rys. 3.4), jeżeli najlepszy potomek będzie każdorazowo znajdował się na początku listy.

Przystąpimy teraz do analizy złożoności obliczeniowej tego drugiego — optymistycznego — przypadku. Rezultat asymptotyczny, który otrzymamy, reprezentuje poniższe twierdzenie:

Twierdzenie 3.2.1 Niech b i d oznaczają odpowiednio współczynnik rozgałęzienia gry i zadaną maksymalną głębokość drzewa. W przypadku optymistycznym złożoność obliczeniowa algorytmu „przycinanie α-β” jest klasy $O(b^{d/2})$.

Dowód. Zwróćmy uwagę na następujące ważne obserwacje. W trakcie pracy algorytmu „przycinania α-β” znamy albo *dokładną wartość* stanu, albo *ograniczenie* (dolne lub górne) na tę wartość. Aby ustalić dokładną wartość, wystarcza (w przypadku optymistycznym) znajomość: dokładnej wartości jednego dziecka i ograniczeń dla $b-1$ pozostałych dzieci. Aby ustalić ogranicznie, wystarcza (w przypadku optymistycznym) tylko znajomość dokładnej wartości jednego dziecka. Powyższe uwagi obrazuje rys. 3.5. Zdefiniujemy dwie wielkości rekurencyjne wypowiadające



Rys. 3.5: Przykład działania algorytmu „przycinanie α-β” (powtórzony na podstawie rys. 3.4) z zaznaczeniem ograniczeń liczbowych, które w przypadku optymistycznym stają się wiadome po poznaniu dokładnej wartości pierwszego dziecka. (źródło: *opracowanie własne*)

się na temat liczby liści[6], które trzeba odwiedzić:

- R_d — minimalna liczba liści (w drzewie o głębokości d), które trzeba odwiedzić, aby poznać dokładną wartość korzenia tego drzewa,
- S_d — minimalna liczba liści (w drzewie o głębokości d), które trzeba odwiedzić, aby poznać ograniczenie na wartość korzenia tego drzewa.

Brzeg obu tych rekurencji to: $R_0 = S_0 = 1$.

[6] w odróżnieniu od rekurencji (3.2), gdzie początkowa jedynka w zapisie $R_d = 1 + \cdots$ powodowała zliczanie wszystkich stanów (nie tylko liści)

Zgodnie z wcześniejszymi obserwacjami zapisujemy:

$$R_d = R_{d-1} + (b-1)S_{d-1}; \tag{3.4}$$

$$S_d = R_{d-1}. \tag{3.5}$$

Podstawiając (3.5) do (3.4), otrzymujemy:

$$R_d = R_{d-1} + (b-1)R_{d-2}. \tag{3.6}$$

Można sprawdzić, że powyższy wzór (3.6) jest dokładny. Dla przykładu z rys. 3.5 otrzymujemy $R_3 = b^2 + b - 1 = 11$. Faktycznie — aby poznać wartość gry w korzeniu (wynoszącą 7) trzeba odwiedzić 11 liści (czyli stanów odległych od korzenia o 3 poziomy).

Wychodząc od rekurencji (3.6) można oszacować z góry liczbę stanów (dla przypadku optymistycznego) poprzez następujący ciąg przejść:

$$\begin{aligned} R_d &= R_{d-1} + (b-1)R_{d-2} \\ &= R_{d-2} + (b-1)R_{d-3} + (b-1)R_{d-2} \\ &= bR_{d-2} + (b-1)R_{d-3} \\ &< bR_{d-2} + (b-1)R_{d-2} \\ &= (2b-1)R_{d-2} \\ &< 2bR_{d-2}. \end{aligned} \tag{3.7}$$

Ostateczną nierówność można zinterpretować w sposób następujący — *efektywny* współczynnik rozgałęziania co każde 2 poziomy jest mniejszy niż $2b$. A więc dla jednego poziomu jest on mniejszy niż $\sqrt{2b}$ (stosując regułę średniej geometrycznej). Rozwijając tę nierówność, otrzymujemy:

$$\begin{aligned} R_d &< 2bR_{d-2} < (2b)^2 R_{d-4} < (2b)^3 R_{d-6} < \cdots < (2b)^k R_{d-2k} \\ &< (2b)^{d/2} R_{d-2d/2} = (2b)^{d/2} R_0 \ \sim O(b^{d/2}) = O\left(\left(\sqrt{b}\right)^d\right). \end{aligned} \tag{3.8}$$

■

Mówiąc w uproszczeniu, pojawia się tu schemat:

$$O(\underbrace{b \cdot 1 \cdot b \cdot 1 \cdots b \cdot 1}_{d})$$

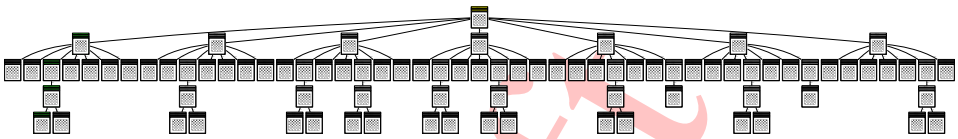
czyli $d/2$-krotne mnożenie współczynnika b. A zatem złożoność $O(b^{d/2})$ wskazuje, że w przypadku optymistycznym algorytm „przycinanie α-β" jest w stanie przeanalizować w tym samym reżimie czasowym dwukrotnie głębsze drzewo niż algorytm min-max, dla którego złożoność obliczeniowa jest klasy $O(b^d)$.

Szacuje się, że w przypadku średnim (biorąc pod uwagę losowe permutacje stanów potomnych) złożoność obliczeniowa „przycinanie α-β" jest rzędu $O\left(b^{3d/4}\right)$.

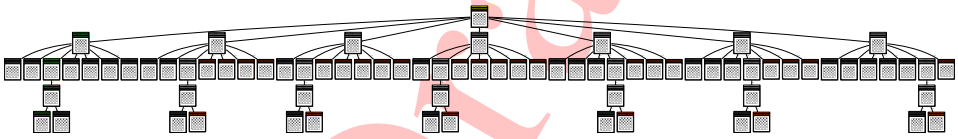
3.2.2 Przykłady działania min-max i „przycinania α-β" dla warcabów

Początkowe fragmenty drzewa gry

Rysunki 3.6–3.9 obrazują początkowe fragmenty drzewa gry w warcaby wygenerowane przez algorytmy min-max i „przycinanie α-β" wyposażone dodatkowo w technikę Quiescence. Zachęcamy czytelnika do powiększania rysunków.

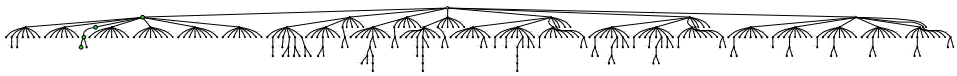


Rys. 3.6: Algorytm *min-max + Quiescence*: zadana głębokość (dla pozycji cichych) 1.0, wygenerowanych stanów 86. (źródło: *opracowanie własne*)



Rys. 3.7: Algorytm *„przycinanie α-β" + Quiescence*: zadana głębokość (dla pozycji cichych) 1.0, wygenerowanych stanów 78. (źródło: *opracowanie własne*)

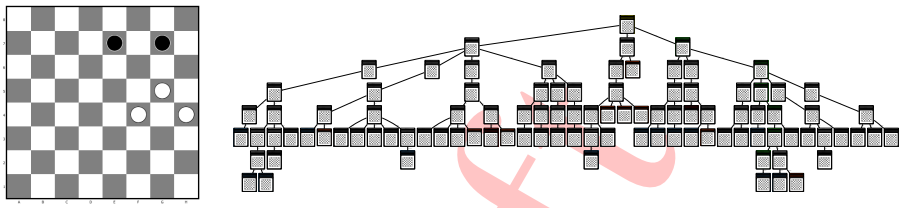
Rys. 3.8: Algorytm *min-max + Quiescence*: zadana głębokość (dla pozycji cichych) 1.5, wygenerowanych stanów 693. (źródło: *opracowanie własne*)



Rys. 3.9: Algorytm *„przycinanie α-β" + Quiescence*: zadana głębokość (dla pozycji cichych) 1.5, wygenerowanych stanów 323. (źródło: *opracowanie własne*)

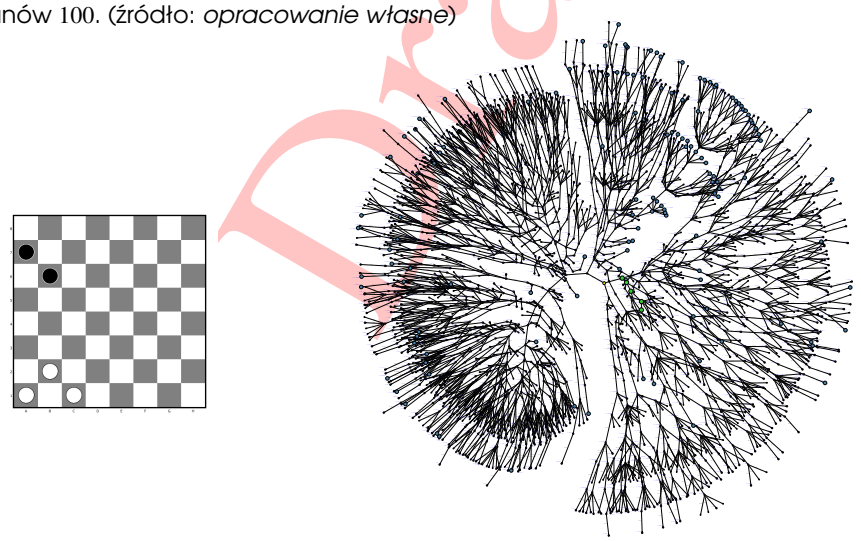
Końcówki warcabowe

Rysunki 3.10–3.13 przedstawiają przykłady końcówek warcabowych przeanalizowanych algorytmem „przycinanie α-β" z włączonym Quiescence. Na każdym rysunku kolorem niebieskim oznaczono pozycje zwycięskie. Kolorem zielonym wyróżniono tzw. *wariant główny* (ang. *principal variation*) — czyli pierwszą napotkaną ścieżkę stanów, która gwarantuje największą wypłatę graczowi rozpoczynającemu przy optymalnym postępowaniu obu graczy.

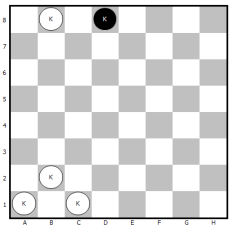


Wariant główny: $(G5 : H6, G7 : F6, F4 : G5, F6 : E5, G5 : F6, E5 : G7, H6 : F8 : D6)$.

Rys. 3.10: Końcówka warcabowa: białe rozpoczynają i wygrywają w 4 posunięciach. Algorytm *„przycinanie α-β" + Quiescence*, zadana głębokość 2.5, wygenerowanych stanów 100. (źródło: *opracowanie własne*)

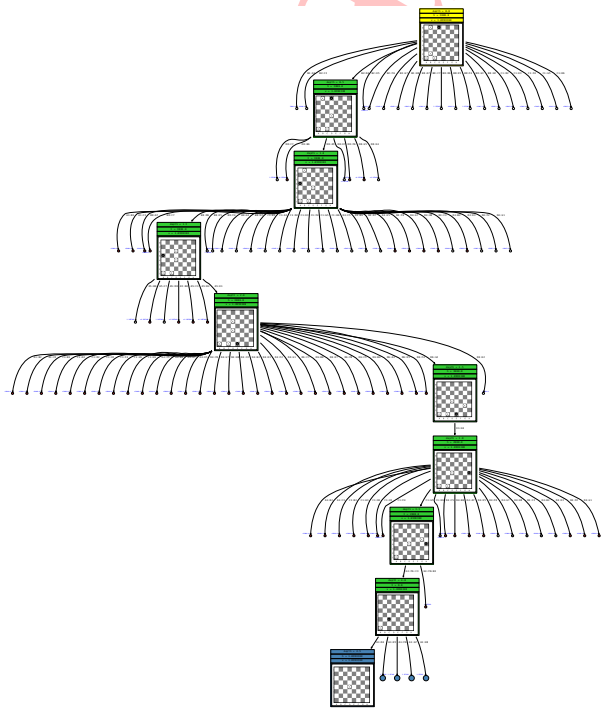


Rys. 3.11: Końcówka warcabowa: kto wygra? Algorytm *„przycinanie α-β" + Quiescence*, zadana głębokość 5.5, wygenerowanych stanów 2845. (źródło: *opracowanie własne*)



```
Searching with sac.game.AlphaBetaPruning...
Searching done. Time: 1789 ms.
Closed states: 54898
General depth limit: 3.5
Maximum depth reached (Quiescence): 4.5
Transposition table size: 52967
Transposition table uses: 69365
Refutation table size: 4611
Refutation table uses: 0
Moves scores: {B2:D4=1.0985902490825263E308, B2:A3=3000.0}
Best move: B2:D4
Principal variation: [B2:D4, D8:A5, B8:D6, A5:E1, D6:G3,
    E1:H4, C1:G5, H4:F6:C3, A1:D4]
```

Rys. 3.12: Końcówka warcabowa: „4 damki vs 1 damka“. Algorytm *„przycinanie α-β“ + Quiescence*, zadana głębokość 3.5, wygenerowanych stanów 54898. (źródło: *opracowanie własne*)



Rys. 3.13: Ilustracja wariantu głównego dla końcówki „4 damki vs 1 damka“ pochodzącego z rys. 3.12. (źródło: *opracowanie własne*)

3.3 Ćwiczenia laboratoryjne (Java + biblioteka *SaC*)

Ćwiczenie 3.1 **Napisz grę Connect 4 (czwórki) pozwalającą na rozgrywkę pomiędzy człowiekiem a sztuczną inteligencją.** Rozpocznij od napisania klasy reprezentującej stan gry Connect 4 (nie skupiając się na przeszukiwaniu minimaksowym), dziedziczącą po `sac.game.GameStateImpl`. Wskazówki: wprowadź stałe lub typ wyliczeniowy na symbole żetnów / pionów graczy tj. `X`, `O`, i symbol pusty (brak zajętości danej komórki planszy); wprowadź stałe określające rozmiar planszy (liczba wierszy × liczba kolumn), zakładając, że są one nie wieksze niż 10, tak aby w największym przypadku kolumny (i tym samym możliwe ruchy) mogły być ponumerowane od 0 do 9; przygotuj konstruktor i konstruktor kopiujący; napisz metodę `toString()`, starając się o czytelny dla gracza wypis planszy, wygodny do prowadzenia rozgrywki; przygotuj metodę wykonującą na planszy pojedynczy ruch (nazwa wg własnego uznania), czyli wrzucenie żetonu do wskazanej kolumny (uwaga: dzięki dziedziczeniu z `sac.game.GameStateImpl` masz dostęp do flagi logicznej informującej, o tym na kogo przypada teraz kolej ruchu — metody `isMaximizingTurnNow()` i `setMaximizingTurnNow()`); kolej ruchu dobrze jest zmieniać tuż przed zakończeniem metody wykonującej pojedynczy ruch; zaimplementuj metodę `hashCode()` (pozwoli to bibliotece *SaC* działać szybciej poprzez używanie gotowych ocen stanów już odwiedzonych); zaimplementuj metodę `generateChildren()` i nie zapomnij o nadaniu nazw stanom potomnym poprzez `setMoveName(...)` — dzięki temu algorytm bedzie mógł przypisać później oceny ruchom; przemyśl i zaimplementuj wg własnych pomysłów i inwencji twórczej metodę oceniającą heurystycznie pozycję (stan) gry Connect 4 — podpięcie metody poprzez mechanizm `setHFunction(...)` — metoda ta będzie używana przez algorytm przeszukujący wtedy, gdy drzewo osiągnie poziom liści, można np. uwzględnić w niej: sumę długości podciągów w różnych kierunkach, preferowanie centrum niż boków lub odwrotnie itp.; uwaga: w pierwszej kolejności upewnij się, że Twoja metoda z funkcją oceny zwraca odpowiednie „nieskończoności” (`Double.POSITIVE_INFINITY` lub `Double.NEGATIVE_INFINITY`) dla stanów zwycięskich; napisz odpowiednią funkcję `main(...)` pozwalającą na prowadzenie rozgrywki z konsoli pomiędzy człowiekiem a sztuczną inteligencją — główna pętla grająca; zapewnij przełącznik pozwalający na rozpoczęcie dowolnemu z graczy; w czasie rozgrywki wyświetlaj dla informacji oceny ruchów wskazane przez algorytm — metoda `getMovesScores()` wywołana na rzecz obiektu stanowiącego algorytm przeszukujący, np. na rzecz obiektu `AlphaBetaPruning`; uwaga: zwracane oceny ruchów będą liczbami zgodnymi z Twoją heurystyczną funkcją oceny, przy czym „nieskończoności” mogą zostać automatycznie przeliczone przez silnik *SaC* na wartości rzędu 10^{308} w typie `double`; wykonuj ruch o najlepszej ocenie w imieniu sztucznej inteligencji — ze zbioru ocen ruchów wychwytuje go gotowa metoda `getFirstBestMove()`; do sczytywania ruchu wybranego przez człowieka z klawiatury należy wykorzystać obiekt `System.in` oraz klasy `java.io.InputStreamReader`, `java.io.BufferedReader` lub alternatywnie klasę `java.util.Scanner`.

Ćwiczenie 3.2 **Przeprowadź eksperymenty „sztuczna inteligencja” vs „sztuczna inteligencja”** Mając do dyspozycji program z ćwiczenia 3.1 przeprowadź kilka eksperymentalnych rozgrywek, w których sztuczne inteligencje o różnych funkcjach oceny

i / lub różnych głębokościach (horyzontach) przeszukiwania będą rywalizowały między sobą. Mecze można rozgrywać uruchamiając np. dwa procesy tego samego programu i przekazując ruchy poprzez konsolę. W szczególności rywalizować mogą np. programy (czyli heurystyki) pochodzące od różnych autorów. Głębokość przeszukiwań można zmieniać za pomocą obiektu `GameSearchConfigurator`.

E **Ćwiczenie 3.3 Rozszerz sztuczną inteligencję do gry Connect 4 o „zmysł cichości”** Mając do dyspozycji program z ćwiczenia 3.1 rozszerz go o możliwość lokalnego pogłębiania horyzontu i przeszukiwania tzw. stanów głośnych zgodnie z techniką *Quiescence*. W ramach biblioteki *SaC* powyższe zadanie sprowadza się do zaimplementowania metody `isQuiet()` na rzecz obiektu dziedziczącego z `sac.game.GameStateImpl` czyli w naszym przypadku stanu gry Connect 4. Metoda ta domyślnie zwraca zawsze wartość `true`. Zastanów się, w jaki sposób można zdefiniować głośność / cichość stanu w tej grze. Wskazówka: pomysł podstawowy może polegać na obserwowaniu procentowej zmiany wartości heurystyki pomiędzy stanami rodzicem i dzieckiem. Sprawdź, czy wprowadzona modyfikacja poprawia jakość gry Twojej sztucznej inteligencji.

3.4 Ćwiczenia laboratoryjne (C# + biblioteka *AISearch*)

E **Ćwiczenie 3.4 Napisz grę Connect 4 (czwórki) pozwalającą na rozgrywkę pomiędzy człowiekiem a sztuczną inteligencją.** Napisz program pozwalający na rozgrywkę w konsoli pomiędzy człowiekiem a sztuczną inteligencją. Zapewnij przełącznik pozwalający na rozpoczęcie dowolnemu graczowi oraz możliwość ustawienia głębokości przeszukiwania drzewa przed rozpoczęciem rozgrywki. W kodzie powinna istnieć łatwa możliwość zmiany rozmiaru planszy. W czasie gry należy wyświetlać na ekran heurystyczne oceny ruchów. Interakcja gracza z programem może polegać na wyborze cyfr 1–9 identyfikujących numer kolumny, w której gracz będzie wstawiał swojego pionka. Struktura plików oraz sposób implementacji jest analogiczny do ćwiczenia 2.10 i 2.15. Utwórz dwie klasy potomne `Connect4State.cs` i `Connect4Search.cs` dziedziczące odpowiednio po klasach bazowych `State.cs` i `AlphaBetaSearch.cs`. Szczegółowe wskazówki implementacyjne zamieszczono w dodatku 11.3.

E **Ćwiczenie 3.5 Zmodyfikuj program z ćwiczenia 3.4 pozwalający na grę na różnych poziomach**. Rozgrywkę na różnych poziomach można zaimplementować m.in. poprzez: wybierania stanów z właściwości `MoveMiniMaxes`, które nie są najlepsze, modyfikacje funkcji heurystycznej. Sama zmiana głębokości przeszukiwania jest zbyt trywialna, aby uznać ją za poprawne wykonanie ćwiczenia.

3.5 Ćwiczenia laboratoryjne (C++ + biblioteka *SI++*)

E **Ćwiczenie 3.6** **Napisz grę Connect 4 (czwórki) pozwalającą na rozgrywkę pomiędzy człowiekiem a sztuczną inteligencją.** Wskazówki: stwórz nową klasę reprezentującą stan planszy (rozmiar nie musi być parametryzowany — można na sztywno ustawić 6 wierszy i 7 kolumn), dziedziczącą po `game_state`, wskazując w szablonie typ reprezentujący ruch (np. `int`); dostarcz implementacje metod: `clone()`, `generate_moves()` (metoda ma zwracać tablicę dopuszczalnych posunięć), `make_move()` (metoda ma zwracać nowy stan odpowiadający wykonanemu posunięciu), `to_string()`, `hash_code()`, `get_h()` (metoda powinna zwracać heurystyczną ocenę bieżącego stanu), `is_terminal()` (metoda ma zwracać $+\infty$ w przypadku wygranej pierwszego gracza[7], $-\infty$ w przypadku wygranej drugiego gracza, 0 w przypadku remisu lub `{}` w przeciwnym wypadku), `is_equal()`; przygotuj funkcję pozwalającą na rozgrywkę z komputerem (grę zaczyna albo człowiek, albo maszyna), w której stworzysz obiekt klasy `game_searcher` z zadaną głębokością przeszukiwania oraz informacją, czy wynik ma być maksymalizowany (drugi argument konstruktora ustawiony na `true`) lub minimalizowany (`false`); wywołanie metody `do_search()` z zadanym stanem początkowym spowoduje wykonanie przeszukiwania — wynik można odebrać za pomocą wywołania metody `get_scores()`, która zwróci tablicę z ocenami poszczególnych ruchów; jeśli gracz jest graczem maksymalizującym, wybierz ruch o najwyższej ocenie i go wykonaj, odbierając nowy stan; w międzyczasie wyświetlaj stan planszy oraz oceny ruchów.

E **Ćwiczenie 3.7** **Zmodyfikuj program z ćwiczenia 3.6, dodając element losowości.** Wybieranie za każdym razem pierwszego najlepszego ruchu prowadzi do monotonii rozgrywki. Aby temu zapobiec, wykonaj dowolny ruch spośród tych, których względna różnica między najlepszym ruchem jest nieduża (np. mniejsza niż 5%).

[7] W C++: `return std::numeric_limits<double>::infinity();`

II Uczenie maszynowe

4. Perceptrony

4.1 Perceptron prosty

Pomysłodawcą pierwszego układu „uczącego się” był Frank Rosenblatt, który w artykule [Ros58] z 1958 r. zaproponował tzw. *perceptron prosty*, naśladujący w uproszczony sposób pracę pojedynczego neuronu w ludzkim mózgu. Zamiarem Rosenblatta była próba pewnego przybliżenia procesu uczenia się na podstawie obserwowanych przykładów, który jest typowy dla ludzi. Rosenblatt był psychologiem i neurobiologiem, a formalny dowód zbieżności dla zaproponowanego przezeń algorytmu dostarczył w 1962 r. Novikoff [Nov62]. Odkrycia te zapoczątkowały dalszy rozwój dziedziny sztucznych sieci neuronowych i ogólniej uczenia maszynowego.

W sensie matematycznym, w wyniku działania algorytmu perceptronu prostego jako rezultat otrzymujemy ***klasyfikator binarny***, czyli funkcję, która przyporządkowuje obiekty wejściowe do jednych z dwóch klas. Przykłady zadania klasyfikacji binarnej to m.in.: filtracja poczty elektronicznej (klasy: spam vs. nie-spam), diagnostyka medyczna (klasy: chory vs. zdrowy), wykrywanie obiektów na obrazach cyfrowych (np. klasy: twarz vs. nie-twarz), itp. Dodatkowo, otrzymany dzięki perceptronowi prostemu klasyfikator jest *liniowy*, co wynika z przyjętego przez Rosenblatta modelu matematycznego. Funkcja obliczająca wartość odpowiedzi perceptronu (przed momentem tzw. progowania) jest funkcją liniową. „Wiąże” ona

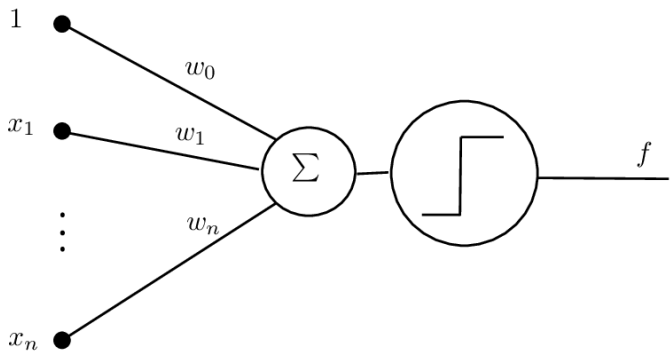
w postaci sumy ważonej własności obiektów wejściowych wyrażone liczbowo (tzw. cechy lub atrybuty obiektów) z pewnymi współczynnikami, które perceptron ma odpowiednio dobrać w procesie uczenia.

Należy zaznaczyć, że zadanie klasyfikacji (jako jedno z zadań uczenia maszynowego) jest tzw. zadaniem *uczenia z nadzorem*[1] (ang. *supervised learning*). Oznacza to, że danymi wejściowymi dla algorytmu uczącego są *pary* informacji — wektor cech i etykieta klasy — opisujące poszczególne obiekty. Na przykład, w zadaniu filtracji poczty elektronicznej jako cechy używane mogą być częstości występowania w wiadomości pewnych kluczowych słów lub wyrażeń (m.in.:„nagroda", „oferta", „darmowy", „karta kredytowa", itp.), zaś klasy są reprezentowane przez etykiety: „spam" i „nie-spam". Nadzór polega na tym, że odpowiednia etykieta musi być dostarczona wraz z każdym obiektem (tu: wiadomością mailową) wchodzącym w skład zbioru uczącego. Innymi słowy, materiał uczący to historyczne przykłady oznakowane przez eksperta z danej dziedziny. Podobnie jak ucząc małe dziecko rozróżniania zwierzątek, pokazujemy palcem przykłady, mówiąc „to kotek", „to ptaszek", itd., tak ucząc klasyfikator antyspamowy, dostarczamy do algorytmu odpowiednio duży zbiór przykładowych wiadomości poczty elektronicznej, „mówiąc": „ta wiadomość to spam", „ta wiadomość to nie-spam". Z kolei zadaniem samego algorytmu uczącego jest wychwycenie (na podstawie zgromadzonych danych) pewnych prawidłowości i wzorców dla każdej z klas (tkwiących w zaobserwowanych cechach) i wykonanie pewnego matematycznego przybliżenia (zakodowania) tychże wzorców. Dzięki temu wynikowy klasyfikator nabiera zdolności do uogólniania (generalizacji), czyli zdolności do skutecznego przyporządkowywania do klas nowych przykładów, niewidzianych w materiale uczącym.

4.1.1 Schemat graficzny

Obliczenia w perceptronie prostym są realizowane zgodnie ze schematem przedstawionym na rys. 4.1 w kierunku od lewej do prawej. Schemat ten z pewną dozą dobrej woli można także traktować jako uproszczony model ludzkiego neuronu. Sygnały wejściowe oznaczone jako $x_1, \ldots, x_n$ reprezentują zaobserwowane lub zmierzone cechy pewnego obiektu. Sygnały te są mnożone przez odpowiednie współczynniki wagowe, a następnie sumowane (co w biologicznych neuronach odbywa się za pomocą tzw. połączeń synaptycznych). Obliczona suma wpływa na sygnał wyjściowy, oznaczony jako f, czyli odpowiedź układu. W związku z klasyfikacją binarną, możliwe są tylko dwa stany odpowiedzi, a jako popularną konwencję przyjmuje się wartości $\{-1, 1\}$. Jeżeli obliczona suma jest powyżej pewnego ustalonego progu, to układ odpowiada wartością 1, w przeciwnym razie wartością -1. Ten element obliczeń nazywany jest w ogólności *funkcją aktywacji*

[1] Lub inaczej: z nauczycielem.



Rys. 4.1: Schemat graficzny perceptronu prostego (źródło: *opracowanie własne*).

neuronu, a w przypadku perceptronu prostego jest to funkcja schodkowa (zaznaczona symbolicznie na wykresie). Myśląc ponownie o biologicznych neuronach, można powiedzieć, że działanie funkcji aktywacji odzwierciedla sposób wzbudzania odpowiedzi neuronu.

Przyjmując jako próg decyzyjny wartość 0, omówiony schemat obliczeń można zapisać następująco:

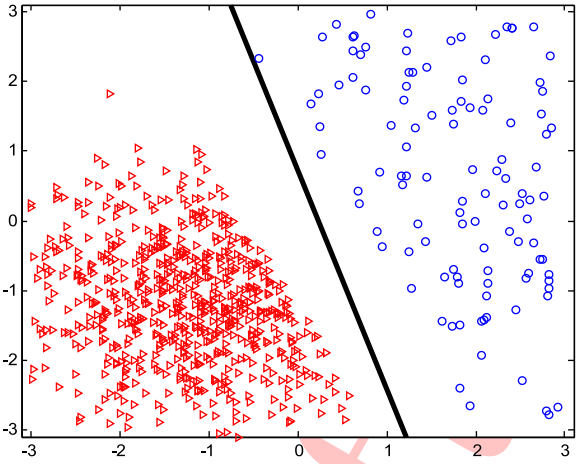
$$s = w_0 + w_1 x_1 + \cdots + w_n x_n. \tag{4.1}$$

$$f(s) = \begin{cases} 1, & \text{dla } s > 0; \\ -1, & \text{dla } s \leqslant 0. \end{cases} \tag{4.2}$$

4.1.2 Notacja, dane, sens geometryczny

Niech $D = \{(\mathbf{x}_i, y_i)\}_{i=1,\ldots,m}$ oznacza zbiór danych uczących (przykładów) zawierający pary, w których $\mathbf{x}_i = (x_{i1}, x_{i2}, \ldots, x_{in}) \in \mathbb{R}^n$ są wektorami cech rzeczywistoliczbowych, a $y_i \in \{-1, 1\}$ skojarzonymi z nimi etykietami klas. A zatem w przypadku perceptronu prostego o przykładach uczących można myśleć jako o punktach w przestrzeni $\mathbb{R}^n$ pokolorowanych za pomocą dwóch kolorów.[2] Ilustrację dla takiej interpretacji stanowi rys. 4.2, gdzie pokazano przykładową klasyfikację binarną na płaszczyźnie ($n = 2$). W tym przypadku każdy przykład (lub inaczej — punkt danych) posiada dwie cechy rzeczywiste, które możemy traktować jak współrzędne kartezjańskie. Wskazana na rysunku czarna prosta stanowi liniową granicę decyzyjną pomiędzy dwiema klasami punktów. Oczywiście jest to prosta przykładowa,

[2] Perceptron prosty wymaga cech rzeczywistoliczbowych. Jest to ograniczenie oznaczające niemożność pracy na zmiennych (cechach) wyliczeniowych, takich jak np. kolor oczu. Z drugiej strony właśnie ten wymóg pozwala myśleć o przykładach uczących i testowych jak o punktach w przestrzeni $\mathbb{R}^n$.



Rys. 4.2: Przykład klasyfikacji binarnej na płaszczyźnie (źródło: *opracowanie własne*).

a w pokazanej sytuacji można wskazać wiele innych prostych (nieskończenie wiele) prawidłowo separujących dane.

Równanie prostej można zapisać jako $w_0 + w_1x_1 + w_2x_2 = 0$. W przypadku $n = 3$, tzn. gdy dane przebywają w trójwymiarowej przestrzeni cech, liniową granicą decyzyjną byłaby pewna płaszczyzna o równaniu: $w_0 + w_1x_1 + w_2x_2 + w_3x_3 = 0$. A zatem w ogólności można powiedzieć, że zadaniem perceptronu prostego jest znalezienie odpowiednich współczynników wielowymiarowej płaszczyzny w przestrzeni $\mathbb{R}^n$ (nazywanej także *hiperpłaszczyzną*) o równaniu:

$$w_0 + w_1x_1 + w_2x_2 + \cdots + w_nx_n = 0. \tag{4.3}$$

Uściślając, chcemy znaleźć pewien wektor współczynników $\mathbf{w} = (w_0, w_1, \ldots, w_n)$, definiujący konkretne równanie płaszczyzny, która właściwie separuje dane uczące wedle ich klasy, tj. taki wektor, że:

$$\forall i, y_i = 1 \quad w_0 + w_1x_{i1} + w_2x_{i2} + \cdots + w_nx_{in} > 0. \tag{4.4}$$

i

$$\forall i, y_i = -1 \quad w_0 + w_1x_{i1} + w_2x_{i2} + \cdots + w_nx_{in} \leqslant 0. \tag{4.5}$$

W nomenklaturze sztucznych sieci neuronowych poszukiwane współczynniki nazywane są często współczynnikami wagowymi lub krótko *wagami*.

Patrząc ponownie na schemat graficzny (rys. 4.1) pewnego komentarza wymaga obecność sygnału wejściowego ustalonego na 1 powiązanego ze współczynnikiem w_0. Oczywiście stała wartość 1 nie stanowi żadnej istotnej informacji wejściowej, która w jakikolwiek sposób różnicowałaby obiekty. Należy jednak zrozumieć, że tym sposobem wprowadzamy współczynnik w_0, który stanowi wyraz wolny w równaniu płaszczyzny. Bez obecności tego wyrazu do konkurencji wchodziłyby tylko te płaszczyzny, które przebiegają przez środek przyjętego układu współrzędnych. Tym samym nie wszystkie zbiory danych, dla których istnieje granica liniowa pomiędzy klasami, mogłyby zostać skutecznie odseparowane. A zatem obecność wyrazu wolnego w_0 jest podyktowana geometrycznie.

Wprowadzimy teraz notację wektorów $\mathbf{x}_i$ rozszerzoną o dodatkową cechę $x_0 = 1$ (dla wygody dalszych zapisów matematycznych):

$$\mathbf{x}_i = (1, x_{i1}, x_{i2}, \ldots, x_{in}).$$

Dzięki temu będziemy mogli iloczyn skalarny wektora wag $\mathbf{w}$ i wektora cech $\mathbf{x}_i$ zapisać krótko jako parę: $\langle \mathbf{w}, \mathbf{x}_i \rangle$:

$$\langle \mathbf{w}, \mathbf{x}_i \rangle = \sum_{j=0}^{n} w_j x_{ij}. \tag{4.6}$$

4.1.3 Algorytm uczenia on-line dla perceptronu prostego

Podany poniżej algorytm 10, nazywany zwyczajowo „regułą perceptronu”[3], przedstawia sposób uczenia on-line dla perceptronu prostego. Uczenie on-line oznacza w

Algorytm 10 „Reguła perceptronu”

1: **procedura** PERCEPTRONLEARNINGRULE(D, η)
2: $\mathbf{w}(0) := (0, 0, \ldots, 0)$ ▷ początkowy wektor wag
3: $k := 0$ ▷ licznik kroków
4: **dopóki** zbiór błędnie sklasyfikowanych punktów $E = \{(\mathbf{x}_i, y_i) \colon y_i \neq f(\langle \mathbf{w}(k), \mathbf{x}_i \rangle)\}$ jest niepusty **wykonaj**
5: wylosuj ze zbioru E dowolną parę $(\mathbf{x}_i, y_i)$
6: popraw wektor wag wg wzoru: $\mathbf{w}(k+1) := \mathbf{w}(k) + \eta y_i \mathbf{x}_i$
7: $k := k + 1$
8: **zwróć** $\mathbf{w}(k)$

ogólności, że poprawki współczynników wagowych odbywają się na bieżąco na podstawie właśnie obejrzanego pojedynczego przykładu[4]. Oprócz zbioru danych

[3] Lub alternatywnie „regułą delty” w przypadku nieco innej formy zapisu algorytmu.

[4] W odróżnieniu, uczenie off-line (stosowane czasami w bardziej zaawansowanych sieciach neuronowych) dokonuje pojedynczej poprawki po obejrzeniu całego zbioru uczącego.

uczących D dodatkowym argumentem podawanym przez użytkownika na wejście algorytmu jest liczba $\eta \in (0,1]$ nazywana *współczynnikiem uczenia* (ang. *learning rate* lub *learning coefficient*). Współczynnik ten decyduje o wielkości dokonywanych poprawek i stanowi on analogię do długości kroku w metodach optymalizacji gradientowej. Oznaczenie $\mathbf{w}(k)$, używane w zapisie algorytmu, oznacza wektor wag w k-tym kroku (lub inaczej po wykonaniu k poprawek). Licznik k pełni rolę czysto informacyjną i będzie rozważany podczas analizy zbieżności algorytmu. W implementacji może zostać pominięty.

Najważniejszą rolę w algorytmie odgrywa krok poprawiania (aktualizacji) wektora wag zgodnie ze wzorem:

$$\mathbf{w}(k+1) := \mathbf{w}(k) + \eta y_i \mathbf{x}_i. \tag{4.7}$$

Postać tego wzoru może być dla czytelnika na ten moment niejasna. Dlaczego składnik poprawki wynosi akurat $\eta y_i \mathbf{x}_i$? Poniżej podamy pewną uproszczoną motywację stojącą za tym wzorem, natomiast jego poprawność stanie się w pełni jasna po analizie zbieżności algorytmu.

Przypatrzmy się przez chwilę następującemu wyrażeniu: $y_i \langle \mathbf{w}(k), \mathbf{x}_i \rangle$. Jeżeli punkt danych $(\mathbf{x}_i, y_i)$ jest aktualnie błędnie klasyfikowany, to na pewno zachodzi nierówność $y_i \langle \mathbf{w}(k), \mathbf{x}_i \rangle \leqslant 0$. Mówiąc dokładniej, jeżeli y_i różni się od wyjścia perceptronu f, to rozważane wyrażenie jest iloczynem dwóch czynników przeciwnych znaków (lub wynosi zero tylko w przypadku, gdy $y_i = 1$ i $\langle \mathbf{w}(k), \mathbf{x}_i \rangle = 0$, co powoduje $f(0) = -1$). Można zatem skonstruować następującą wielkość reprezentującą błąd dla i-tego przykładu:

$$e_i = -y_i \langle \mathbf{w}(k), \mathbf{x}_i \rangle. \tag{4.8}$$

Gdy wyjście perceptronu jest niezgodne z oczekiwaną etykietą klasy, to $e_i > 0$ — błąd jest obecny. Gdy zaś wyjście perceptronu jest zgodne z oczekiwaną etykietą klasy, to $e_i \leqslant 0$, co można interpretować jako brak błędu (lub umownie jako błąd ujemny). Podstawowa technika znana z metod optymalizacji, nakazuje wyznaczyć gradient (czyli wektor pochodnych cząstkowych funkcji błędu ze względu na poszczególne parametry) i poprawiać optymalizowany wektor parametrów w kierunku przeciwnym do gradientu, tzn.:

$$\mathbf{w}(k+1) := \mathbf{w}(k) - \frac{\partial e_i}{\partial \mathbf{w}(k)}. \tag{4.9}$$

Łatwo sprawdzić, że

$$\frac{\partial e_i}{\partial \mathbf{w}(k)} = -y_i \mathbf{x}_i, \tag{4.10}$$

co uzasadnia postać wzoru (4.7) na rzecz i-tego punktu danych. Jest to jednak uzasadnienie tylko częściowe. Oznacza tylko bowiem to, że konsekwentne stosowanie tego wzoru dla pojedynczego ustalonego punktu danych w pewnym momencie spowoduje, że punkt ten będzie już dobrze sklasyfikowany. Wyjaśnienie to nie pozwala jednak wnioskować, że *wszystkie* punkty danych zostaną w pewnym momencie prawidłowo sklasyfikowane w konsekwencji uczenia on-line reprezentowanego przez Algorytm 10, czyli w efekcie iteracyjnego wykonywania wzoru (4.7) na rzecz różnych punktów danych. A zatem należy zastanowić się, czy możliwe jest pojawienie się oscylacji w poprawkach i brak zbieżności algorytmu. Twierdzenie przedstawione w kolejnym rozdziale i jego dowód zaprzeczą takiej możliwości.

4.1.4 Twierdzenie o zbieżności algorytmu uczącego

Dla ścisłego wypowiedzenia twierdzenia o zbieżności potrzebna będzie formalna definicja *liniowej separowalności* danych:

Definicja 4.1.1 — liniowa separowlaność. Mówimy, że zbiór danych $D = \{(\mathbf{x}_i, y_i)\}_{i=1,\ldots,m}$ jest liniowo separowalny wtedy i tylko wtedy, gdy istnieje pewien wektor $\mathbf{w}^* = (w_0^*, w_1^*, \ldots, w_n^*)$, taki że:

$$\forall i, y_i = 1 \quad \langle \mathbf{w}^*, \mathbf{x}_i \rangle > 0, \tag{4.11}$$

$$\forall i, y_i = -1 \quad \langle \mathbf{w}^*, \mathbf{x}_i \rangle \leqslant 0. \tag{4.12}$$

Twierdzenie 4.1.1 Jeżeli zbiór danych uczących jest liniowo separowalny, to algorytm perceptronu prostego zatrzyma się po skończonej liczbie kroków, ograniczonej z góry w następujący sposób:

$$k \leqslant \frac{R^2}{\gamma^2},$$

gdzie R, γ to pewne dodatnie stałe określone przez rozkład punktów danych.

Dowód. Wprowadźmy następujące oznaczenia. Niech $\mathbf{w}^*$ oznacza dowolny optymalny wektor wag (tzn. wektor współczynników płaszczyzny optymalnie separującej dane). Nie znamy z góry zawartości takiego wektora, ale wiemy, że takowy istnieje, skoro zbiór danych jest liniowo separowalny. Bez straty ogólności przyjmijmy, że wektor ten jest unormowany do długości 1, tzn. $\|\mathbf{w}^*\| = 1$. Niech $R > 0$ oznacza tzw. *promień danych*, zdefiniowany jako

$$R = \max_{i=1,\ldots,m} \|\mathbf{x}_i\|. \tag{4.13}$$

Innymi słowy, dane można zamknąć w kuli o promieniu R. Dalej, niech $\gamma' > 0$

oznacza tzw. *margines separacji* zdefiniowany jako:

$$\gamma' = \min_{i=1,\ldots,m} \frac{y_i \langle \mathbf{w}^*, \mathbf{x}_i \rangle}{\|(w_1^*, \ldots, w_n^*)\|}. \tag{4.14}$$

Można sprawdzić, że wyrażenie pod operatorem minimum reprezentuje odległość punktu danych $\mathbf{x}_i$ od płaszczyzny separacji określonej wektorem $\mathbf{w}^*$. Dodatkowo wprowadźmy stałą γ będącą wielkością proporcjonalną do γ' z zaniedbaniem mianownika, tj. $\gamma = \gamma' \|(w_1^*, \ldots, w_n^*)\| = \min_{i=1,\ldots,m} y_i \langle \mathbf{w}^*, \mathbf{x}_i \rangle$.

Pokażemy, że kąt pomiędzy wektorem $\mathbf{w}^*$ a kolejnymi wektorami $\mathbf{w}(k)$ stopniowo maleje w trakcie pracy algorytmu. Kosinus tego kąta można przedstawić jako

$$\frac{\langle \mathbf{w}(k), \mathbf{w}^* \rangle}{\|\mathbf{w}(k)\| \underbrace{\|w^*\|}_{1}},$$

a zatem potrzebujemy obserwować zmienność iloczynu skalarnego $\langle \mathbf{w}(k), \mathbf{w}^* \rangle$ oraz normy $\|\mathbf{w}(k)\|$ w trakcie pracy algorytmu. Wykonamy to w sposób rekurencyjny, stosując wielokrotnie wzór (4.7) na poprawkę wag.

Rozwijając rekurencyjnie $\langle \mathbf{w}(k), \mathbf{w}^* \rangle$ otrzymujemy ciąg nierówności:

$$\begin{aligned}
\langle \mathbf{w}^*, \mathbf{w}(k) \rangle &= \langle \mathbf{w}^*, \mathbf{w}(k-1) + \eta y_i \mathbf{x}_i \rangle = \langle \mathbf{w}^*, \mathbf{w}(k-1) \rangle + \eta \underbrace{y_i \langle \mathbf{w}^*, \mathbf{x}_i \rangle}_{\geqslant \gamma} \\
&\geqslant \langle \mathbf{w}^*, \mathbf{w}(k-1) \rangle + \eta\gamma \\
&\geqslant \langle \mathbf{w}^*, \mathbf{w}(k-2) \rangle + 2\eta\gamma \\
&\vdots \\
&\geqslant \underbrace{\langle \mathbf{w}^*, \mathbf{w}(0) \rangle}_{0} + k\eta\gamma.
\end{aligned} \tag{4.15}$$

Nierówność $\langle \mathbf{w}^*, \mathbf{w}(k) \rangle \geqslant k\eta\gamma$ oznacza, że obserwowany iloczn skalarny rośnie podczas pracy algorytmu i po k krokach wynosi przynajmniej $k\eta\gamma$ (pamiętając, że wszystkie te stałe są dodatnie). Wzrost iloczynu skalarnego to warunek konieczny, aby kąt pomiędzy wektorami malał, ale niewystarczający, ponieważ norma $\|\mathbf{w}(k)\|$ może teoretycznie także rosnąć. Pokażemy, że taki wzrost normy może co najwyżej następować w wolniejszym tempie względem indeksu k.

Bezpośrednie rozwinięcie rekurencyjne $\|\mathbf{w}(k)\|$ jest niewygodne ze względu na pierwiastek kwadratowy występujący we wzorze normy. Rozwińmy zatem kwadrat

normy (odkładając operację pierwiastkowania na później):

$$\begin{aligned}\|\mathbf{w}(k)\|^2 = \langle \mathbf{w}(k), \mathbf{w}(k)\rangle &= \langle \mathbf{w}(k-1) + \eta y_i \mathbf{x}_i, \mathbf{w}(k-1) + \eta y_i \mathbf{x}_i\rangle \\ &= \|\mathbf{w}(k-1)\|^2 + \underbrace{2\eta y_i \langle \mathbf{w}(k-1), \mathbf{x}_i\rangle}_{\leqslant 0} + \eta^2 \underbrace{y_i^2}_{1} \underbrace{\|\mathbf{x}_i\|^2}_{\leqslant R^2} \\ &\leqslant \|\mathbf{w}(k-1)\|^2 + \eta^2 R^2 \\ &\leqslant \|\mathbf{w}(k-2)\|^2 + 2\eta^2 R^2 \\ &\vdots \\ &\leqslant \underbrace{\|\mathbf{w}(0)\|^2}_{0} + k\eta^2 R^2. \end{aligned} \tag{4.16}$$

A zatem po k krokach algorytmu $\|\mathbf{w}(k)\|$ wynosi co najwyżej $\sqrt{k}\eta R$.

Otrzymane ograniczenia nierównościowe (4.15) i (4.16) łączymy za pomocą nierówności Cauchy'ego-Schwarza — $\langle a, b\rangle \leqslant \|a\|\|b\|$ — otrzymując:

$$k\eta\gamma \leqslant \langle \mathbf{w}(k), \mathbf{w}^*\rangle \leqslant \|\mathbf{w}(k)\| \underbrace{\|\mathbf{w}^*\|}_{1} \leqslant \sqrt{k}\eta R. \tag{4.17}$$

Można zauważyć, że powyższy układ nierówności może pozostawać prawdziwy tylko dla skończonego zbioru indeksów k, ponieważ dolne ograniczenie skaluje się liniowo wraz z k, zaś górne skaluje się tylko z $\sqrt{k}$. Rozwiązując skrajną nierówność ze względu na k otrzymujemy ostateczne ograniczenie na maksymalną liczbę kroków w algorytmie:

$$\begin{aligned} k\eta\gamma &\leqslant \sqrt{k}\eta R \\ k &\leqslant \frac{R^2}{\gamma^2}. \end{aligned} \tag{4.18}$$

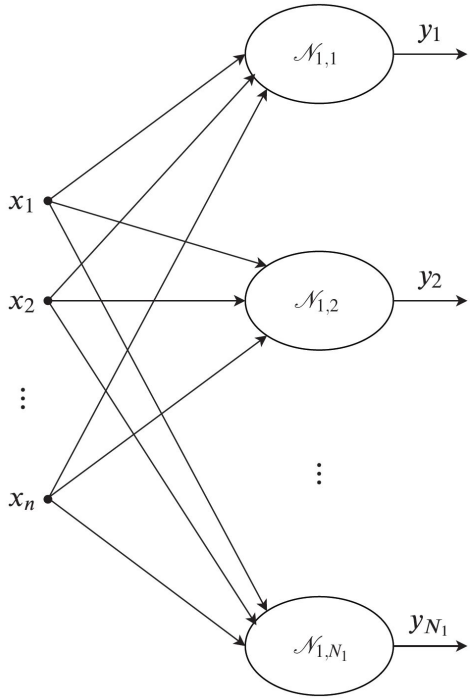
■

4.2 Perceptron wielowarstwowy

4.2.1 Schematy sieci i oznaczenia

Sztuczne neurony mogą tworzyć różnego rodzaju większe struktury zwane sieciami neuronowymi [Mur12; Rut12]. Jedną z prostszych takich struktur jest tzw. sieć jednowarstwowa. Schemat sieci jednowarstwowej prezentuje rys. 4.3. Sieci mogą również posiadać strukturę wielowarstwową, gdzie sygnały są przekazywane z warstwy poprzedniej do kolejnej przy założeniu, że neurony w tej samej warstwie nie są ze sobą połączone, a sieć nie posiada sprzężeń zwrotnych. Są to tak zwane

sieci jednokierunkowe (ang. *feedforward neural networks*). W wielowarstwowych sieciach muszą istnieć przynajmniej dwie warstwy: wejściowa i wyjściowa. Pomiędzy nimi mogą znajdować się warstwy ukryte. W sytuacji kiedy sieć zawiera tylko dwie warstwy, warstwa wejściowa utożsamiana jest z warstwą ukrytą. Niektóre opracowania interpretują wektor sygnałów wejściowych podawanych do sieci neuronowych jako warstwę wejściową. Schemat sieci wielowarstwowej znajduje się na rys. 4.4.

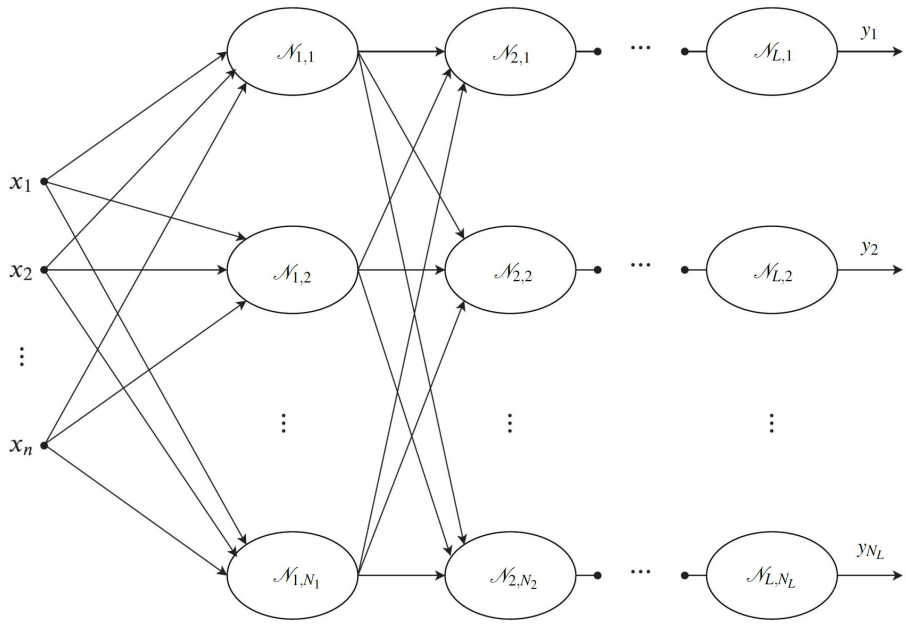


Rys. 4.3: Sieć perceptronowa jednowarstwowa. Użyta ogólna notacja $\mathcal{N}_{l,k}$ oznacza k-ty neuron w warstwie l-tej (źródło: *opracowanie własne*).

Zakładając, że rozpatrujemy sieć neuronową posiadającą L warstw, należy zauważyć, że wyjście neuronu k w warstwie l będzie jednocześnie k-tym wejściem dowolnego neuronu w warstwie $l+1$. Korzystając z tej zależności można zapisać, że $y_{l,k} = x_{l+1,k}$, co uogólniając można przedstawić w postaci zapisu:

$$x_{l,k} = \begin{cases} x_k, & \text{dla } k > 0, l = 1; \text{(sygnały wejściowe pierwszej warstwy);} \\ y_{l+1,k}, & \text{dla } k > 0, l = 2, \ldots, L; \\ 1, & \text{dla } k = 0, l = 1, \ldots, L; \end{cases} \tag{4.19}$$

gdzie przez $x_{l,0}$ rozumiemy wejście progowe, na które zawsze podawana jest war-



Rys. 4.4: Sieć perceptronowa wielowarstwowa (źródło: *opracowanie własne*).

tość 1. Wektor wszystkich wag neuronu k w warstwie l może zapisać jako:

$$\mathbf{w}_{l,k} = \left(w_{l,k,0}, w_{l,k,1}, \ldots, w_{l,k,N_{l-1}}\right), \qquad l = 1,\ldots,L, \quad k = 1,\ldots,N_l, \quad (4.20)$$

gdzie przez N_{l-1} rozumiemy liczbę neuronów w warstwie $l-1$. A zatem przyjęty sposób numeracji pozwala interpretować indeksy wagi $w_{l,k,j}$ skojarzonej z pewnym połączeniem jako kolejno: l — numer warstwy, k — „dokąd" prowadzi połączenie (do którego neuronu w warstwie l), j — „skąd" wychodzi połączenie (z którego neuronu w warstwie $l-1$). Przyjęcie konwencji, gdzie numer „dokąd" poprzedza „skąd" może wydawać się nienaturalne, ale wynika z faktu, że implementacje obliczeń sieci neuronowych są zwykle realizowane macierzowo. Dla każdej warstwy, wagi stojące przy połączeniach prowadzących do tego samego neuronu kolejnej warstwy są przechowywane w tym samym wierszu macierzy, co pozwala na obliczenie wektora sum ważonych danej warstwy poprzez mnożenie macierzowe: macierz wag · kolumna sygnałów wejściowych. A zatem każdy element takiego wynikowego wektora (kolumnowego) jest iloczynem skalarnym odpowiedniego wiersza wag i sygnałów wejściowych — patrz dalej na wzór (4.22).

Sygnał wyjściowy dowolnego neuronu $\mathcal{N}_{l,k}$ jest obliczany jako:

$$y_{l,k} = f(s_{l,k}), \quad (4.21)$$

gdzie f jest przyjętą *funkcją aktywacji neuronu*, a jej argument $s_{l,k}$ jest sumą ważoną sygnałów wejściowych (lub inaczej — iloczynem skalarnym wektora wag i wektora wejść):

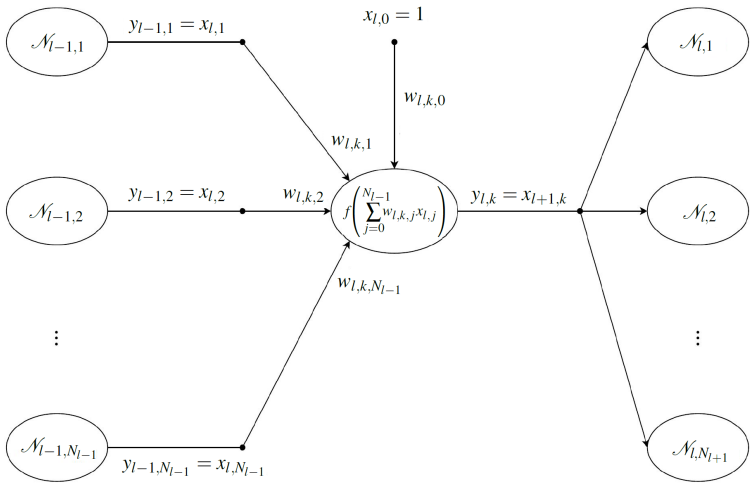
$$s_{l,k} = \sum_{j=0}^{N_{l-1}} w_{l,k,j} \cdot x_{l,j}. \tag{4.22}$$

Przykłady powszechnie stosowanych funkcji aktywacji zostaną podane w kolejnym punkcie.

Schemat działania pojedynczego k-tego neuronu w warstwie l przedstawia rys. 4.5. Należy zauważyć, że sygnały wyjściowe neuronów w warstwie L:

$$y_{L,1}, y_{L,2}, \ldots, y_{L,N_L}, \tag{4.23}$$

są jednocześnie sygnałami wyjściowymi dla całej sieci neuronowej [Pie19].



Rys. 4.5: Schemat działania k-tego neuronu w warstwie l (źródło: *opracowanie własne*).

4.2.2 Uniwersalna aproksymacja

Dla pożądanego działania sieci ważne jest dobranie odpowiedniej liczby jej warstw oraz liczby neuronów. Wielkości te są uzależnione od problemu, który sieć powinna rozwiązywać. Zgodnie z rezultatami matematycznymi Cybenki i Kołmogorowa o aproksymacji za pomocą kombinacji *funkcji sigmoidalnych*, czyli funkcji postaci

$$f(s) = \frac{1}{1+e^{-s}}, \tag{4.24}$$

gdzie s to pewna suma ważona argumentów wejściowych, wiadomy jest fakt, że za pomocą sieci z jedną warstwą ukrytą można przybliżyć dowolną funkcję ciągłą. Twierdzenie Kołmogorowa mówi, że funkcję ciągłą wielu zmiennych można przedstawić w postaci sumy funkcji jednej zmiennej [Kol57]. Z kolei twierdzenie Cybenki mówi, że sigmoidalna sieć neuronowa ma uniwersalne właściwości aproksymacyjne [Cyb89], czyli że dla dowolnej funkcji ciągłej można dobrać odpowiednią sigmoidalną sieć neuronową, która ją przybliży z dowolnie małym błędem. Niestety, powyższe rezultaty są egzystencjalne, a nie konstruktywne. To znaczy, stwierdzają one istnienie wspomnianych własności, jednak nie podają konkretnego przepisu, który mówiłby, w jaki sposób uzyskać właściwy aproksymator dla podanego zbioru danych.

4.2.3 Przeuczenie i zdolność do uogólniania

Zbyt mała liczba neuronów w sieci może skutkować niedostatecznym dopasowaniem utworzonej powierzchni funkcyjnej do prawidłowości tkwiących w danych. Z kolei zbyt duża liczba neuronów przy jednocześnie zbyt małej liczbie przykładów uczących może skutkować *przeuczeniem* sieci (ang. *overfitting*). Przeuczenie to zbyt dokładne dopasowanie powierzchni funkcyjnej sieci do poszczególnych punktów danych uczących i tym samym do szumów tkwiących w nich. Powoduje to zwykle duże lokalne wahania uzyskanej powierzchni funkcyjnej, niezgodne z ogólnymi własnościami modelowanego zjawiska. Mówiąc jeszcze inaczej, sieć przeuczona potrafi bardzo dokładnie odtwarzać dane uczące, ale ma niską dokładność na danych testowych (niewidzianych podczas uczenia), czyli słabą *zdolność do uogólniania*[5]. A właśnie na tej drugiej własności powinno nam zależeć.

4.2.4 Popularne funkcje aktywacji neuronu

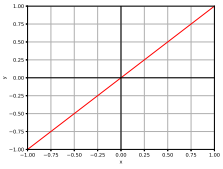
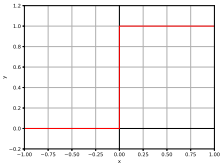
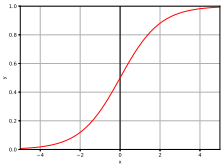
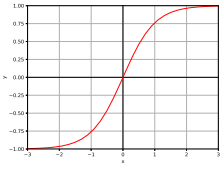
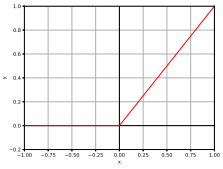
Wspomniana funkcja sigmoidalna a także pokrewny jej tangens hiperboliczny stanowiły historycznie najczęściej stosowane funkcje aktywacji (szczególnie w minionym stuleciu). Współcześnie, największą popularnością cieszy się funkcja o nazwie ReLU (ang. *Rectified Linear Unit*) i pewne jej warianty. Należy zaznaczyć, że mowa tu o aktywacjach dla warstw ukrytych. Jeżeli chodzi o warstwę wyjściową, to wybór funkcji aktywacji dla niej jest zwykle podyktowany typem zadania matematycznego, które sieć realizuje. W przypadku zadania regresji (lub równoważnie aproksymacji), gdzie na wyjściu przewidywana ma być pewna liczba rzeczywista (lub wektor takich liczb) — np. siła wiatru, wartość waluty, natężenie prądu itp. — zwyczajowo nie załącza się żadnej specjalnej funkcji aktywacji, co oznacza równoważnie że działa wówczas funkcja liniowa. W przypadku zadania klasyfikacji, najbardziej popularną jest tzw. funkcja softmax. Jest ona pokrewna

[5] Inaczej: generalizacji.

do funkcji sigmoidalnej, ale wymusza sumowanie się wszystkich aktywacji do jedności, tak jak w rozkładach prawdopodobieństwa, co pozwala na wskazanie wynikowej klasy zgodnie z numerem neuronu o największej wartości aktywacji softmax (prawdopodobieństwo najbliższe wartości 1).

Najczęściej używane funkcje aktywacji wraz z ich pochodnymi zostały przedstawione w tabeli 4.1. Należy zaznaczyć, że pochodne odgrywają ważną rolę podczas uczenia sieci podczas wstecznych przebiegów obliczeń.

Tabela 4.1: Zestawienie wybranych funkcji aktywacji neuronu (źródło: *opracowane na podstawie:* `https://en.wikipedia.org/wiki/Activation_function`).

nazwa funkcji	wykres	wzór	pochodna
liniowa		$f(s) = s$	$f'(s) = 1$
skok jednostkowy		$f(s) = \begin{cases} 0, & \text{dla } s < 0; \\ 1, & \text{dla } s \leqslant 0. \end{cases}$	$f'(s) = 0 \text{ dla } s \neq 0$
sigmoidalna		$f(s) = \frac{1}{1+e^{-s}}$	$f'(s) = f(s)(1-f(s))$
tangens hiperboliczny		$f(s) = \tanh(s) = \frac{e^s - e^{-s}}{e^s + e^{-s}}$	$f'(s) = 1 - f^2(s)$
ReLU		$f(s) = \begin{cases} 0, & \text{dla } s \leqslant 0; \\ s & \text{dla } s > 0. \end{cases}$	$f'(s) = \begin{cases} 0, & \text{dla } s < 0; \\ \text{nie ist.}, & \text{dla } s = 0; \\ 1, & \text{dla } s > 0. \end{cases}$

4.3 Algorytm wstecznej propagacji błędów

Podstawową metodą uczenia jednokierunkowej sieci neuronowej jest ***algorytm wstecznej propagacji błędów*** (ang. *backpropagation*) [Mur12; Rut12] opracowany w 1986 r. przez Davida Rumelharta, Geoffrey'a Hintona i Ronalda Williamsa [RHW86]. Sieć neuronowa podczas obliczania odpowiedzi przepuszcza sygnał wejściowy podany do sieci przez wszystkie warstwy, aż do warstwy wyjściowej. Uczenie sieci odbywa się w przeciwnym kierunku. Algorytm wstecznej propagacji błędu, tak jak sugeruje nazwa, po obliczeniu odpowiedzi sieci koryguje wagi neuronów, propagując błąd wyjściowy „wstecz", z uwzględnieniem połączeń między warstwami, a także funkcji aktywacji neuronów.

Do korygowania parametrów (wag) sieci potrzebna jest pewna funkcja błędu wyrażająca ilościowo, jak duża jest niezgodność pomiędzy pożądaną odpowiedzią a faktycznie otrzymanym wyjściem sieci. Stosowane bywają różne funkcje błędu, a ich wybór jest także powiązany z rodzajem realizowanego zadania. Dla zadania klasyfikacji powszechnie wybieraną do użycia funkcją błędu jest *entropia krzyżowa*, natomiast dla zadania regresji (czyli inaczej aproksymacji) taką funkcją jest *błąd kwadratowy*. Przy okazji można zwrócić tu uwagę na pewną subtelność nazewniczą związaną z tym, że w literaturze stosowana bywa również nazwa *funkcja straty* (ang. *loss function*). I na przykład spotkać można nazwy angielskie: cross-entropy loss, squared loss. Rzecz w tym, że funkcja straty jest zdefiniowana na rzecz pojedynczego przykładu uczącego lub testowego — tj. pojedynczej pary $(\mathbf{x}_i, \mathbf{y}_i)$, natomiast funkcje błędu są zwyczajowo rozumiane jako *sumy* wartości funkcji straty dla większej liczby przykładów, np. dla całego zbioru uczącego (tryb uczenia off-line) lub też pewnego jego losowego podzbioru zwanego także wsadem (tryb uczenia tzw. mini-batchami).

4.3.1 Indukcja wyrażeń błędu i poprawki wag sieci

Dla uproszczenia rozważań w poniższych wyprowadzeniach wybrano kwadratową funkcję straty, i tym samym należy myśleć, że sieć neuronowa realizuje zadanie regresji czyli przewidywanie pewnej wielkości rzeczywistoliczbowej. Zgodnie z ogólną metodą najmniejszych kwadratów, wspomniany błąd pomiędzy odpowiedzią sieci a wyjściem pożądanym jest obliczany jako kwadrat różnicy pomiędzy tymi wielkościami. Należy uświadomić sobie, że zmiana praktycznie dowolnej wagi sieci ma wpływ na ten błąd i poprzez odpowiednią zmianę takiej wagi można błąd redukować. Wzory na poprawki wyznaczone dla wag znajdujących się blisko warstwy wejściowej sieci przyjmują postać długich iloczynów (lub sum iloczynów), a pewne fragmenty tych wzorów związane z końcowymi warstwami powtarzają się. Algorytm wstecznej propagacji błędów (zaproponowany przez Rumelharta i in.) stara się właśnie wykorzystać obecność tych powtórzeń

w celu zredukowania czasu obliczeń, dlatego też algorytm ten jest sformułowany w postaci indukcyjnej.

Przypuśćmy, że ustalony jest pewien przykład uczący jako wektor wejściowy do sieci, a oczekiwany dla niego wektor wyjściowy to $(y_1^*, \ldots, y_{N_L}^*)$. Wówczas błąd sieci neuronowej możemy zdefiniować jako:

$$\frac{1}{2}e^2 = \frac{1}{2}\sum_{k=1}^{N_L}(y_{L,k} - y_k^*)^2 . \tag{4.25}$$

Zgodnie z najprostszym podejściem gradientowym ogólny wzór na korektę pewnej wagi w sieci możemy zapisać następująco:

$$w_{l,k,j}(t+1) = w_{l,k,j}(t) - \eta \frac{\partial \frac{1}{2}e^2}{\partial w_{l,k,j}(t)}. \tag{4.26}$$

Indeks t zapisany w nawiasie oznacza numer kroku w algorytmie.

Poniżej podane zostaną ogólne wzory pozwalające realizować poprawki wag zgodnie z algorytmem wstecznej propagacji błędu, w których pochodną $\partial \frac{1}{2}e^2 / \partial w_{l,k,j}(t)$ oblicza się za pomocą odpowiednio określonych indukcyjnie *wyrażeń błędu* $\delta_{l,k}$. Natomiast w punkcie 4.3.2 czytelnik będzie miał okazję porównać, jak wzory te mają się do wzorów (długich iloczynów pochodnych) wyprowadzanych „ręcznie" na rzecz przykładowej prostej sieci neurnowej z jedną warstwą ukrytą.

Backpropagation — indukcja wyrażeń błędu:

$$\delta_{l,k} := f'(s_{l,k}) \cdot \begin{cases} y_{L,k} - y_k^*, & l = L; \\ \sum_{j=1}^{N_{l+1}} w_{l+1,j,k}\,\delta_{l+1,j}, & l = L-1, L-2, \ldots, 1, \end{cases} \tag{4.27}$$

gdzie $\delta_{l,k}$ oznacza wyrażenie błędu k-tego neuronu w warstwie l.

Backpropagation — poprawka dowolnej wagi w sieci (gradient prosty):

$$w_{l,k,j}(t+1) := w_{l,k,j}(t) - \eta\ \delta_{l,k} x_{l,j}. \tag{4.28}$$

Istotnym jest fakt, że wyrażenia $\delta_{l,k}$ związane z pochodnymi propagowanych błędów są w algorytmie obliczane od prawej strony schematu sieci ku lewej, tj. zgodnie z malejącą kolejnością indeksu $l = L, L-1, \ldots, 1$. Złożenie wzorów (4.27) i (4.28) wynika z dwóch reguł rachunku różniczkowego: *reguły łańcuchowej* (ang. *chain rule*) i *reguły sumy*. Ta pierwsza mówi, że pochodna funkcji zagnieżdżonej jest obliczana jako pochodna funkcji zewnętrznej razy pochodna funkcji wewnętrznej. Ta druga mówi, że pochodna sumy to suma pochodnych. Innymi słowy pochodne cząstkowe funkcji błędu obliczane w sieci neuronowej wzdłuż

pewnej pojedynczej ścieżki połączeń powinny być mnożone, zaś pochodne obliczane wzdłuż ścieżek alternatywnych powinny być sumowane. W szczególności, zasadniczy krok indukcyjny — druga linia (4.27) — warto zapamiętać słownie w sposób następujący: wyrażenie błędu $\delta_{l,k}$ na rzecz k-tego neuronu w warstwie l obliczane jest jako iloczyn pochodnej jego funkcji aktywacji oraz ważonej sumy (wziętej po połączeniach wychodzących) wyrażeń błędu $\delta_{l+1,\cdot}$ neuronów w kolejnej warstwie. Pomimo że w niniejszym punkcie omawiane są tylko sieci perceptronowe powyższy stwierdzenie jest ogólne i prawdziwe dla innych sieci jednokierunkowych z różnymi rodzajami warstw i różnymi funkcjami aktywacji. M.in. jest ono prawdziwe dla ważnych współcześnie głębokich sieci splotowych (ang. *convolutional neural networks*).

Ogólny krok indukcyjny wstecznej propagacji błędu warto zapamiętać jako: $\delta_{l,k}$:= pochodna funkcji aktywacji neuronu $\mathcal{N}_{l,k}$ razy suma ważona (wzięta po połączeniach wychodzących z $\mathcal{N}_{l,k}$) wyrażeń błędu $\delta_{l+1,\cdot}$ neuronów $\mathcal{N}_{l+1,\cdot}$ w kolejnej warstwie.

Zaprezentowany dotychczas rodzaj uczenia to tzw. *uczenie on-line*. Polega ono na modyfikacji wag sieci każdorazowo po „obejrzeniu" pojedynczego przykładu uczącego. Odmiennym rodzajem jest *uczenie off-line*, gdzie poprawka każdej wagi odbywa się dopiero po obejrzeniu przez sieć wszystkich przykładów i wyznaczeniu sumarycznego błędu. Uczenie off-line jest dokładniejsze, ponieważ posługujemy się w nim pełnym gradientem, ale znacznie wolniejsze. W uczeniu on-line poprawki wykonywane są znacznie częściej, ale w danym momencie posługujemy się tak naprawdę tylko pojedynczym składnikiem gradientu ze względu na ustalony przykład wejściowy (spośród wszystkich m składników). Mówi się wówczas o tzw. gradiencie *stochastycznym*, lub inaczej losowym, co oznacza, że proces uczenia, pomimo schodzenia w kierunku antygradientu stochastycznego (SGD — ang. *Stochastic Gradient Descent*) przypomina częściowo losowe błądzenie, ponieważ możemy mieć do czynienia z oscylacjami w trajektorii wektora wag.

Mówiąc ogólnie, uczenie sieci polega na poszukiwaniu minimum funkcji błędu, która jest funkcją wielu zmiennych. Funkcja ta posiada tyle zmiennych, ile jest wag (parametrów) w sieci, a co za tym idzie tak skomplikowana funkcja może posiadać wiele minimów lokalnych. Niestety każdy algorytm gradientowy nie jest odporny na to zjawisko i może się zdarzyć, że utknie w jednym z takich minimów.

W procesie uczenia duże znaczenie ma również współczynnik uczenia η. Niestety nie istnieje ogólna metoda doboru jego wartości. Duża wartość współczynnika powoduje duże zmiany wag i gdy funkcja celu jest stroma, może to skutkować oscylacją wokół rozwiązania. Niska wartość współczynnika uczenia, sprawdzająca się przy stromych funkcjach celu, wydłuża proces uczenia na powierzchniach, gdzie

wartości gradientu są niewielkie. Współczynnik ten można w sposób równoważny interpretować jako wielkość kroku poprawki w metodach gradientowych. Wartość współczynnika zwykle dobiera się eksperymentalnie i zależy ona od problemu, który należy rozwiązać [Pie19].

Istnieje wiele modyfikacji metody wstecznej propagacji błędu, mających na celu lepsze radzenie sobie z problemem minimów lokalnych, a także przyspieszenie procesu uczenia. Niektóre z tych modyfikacji to: uczenie z rozpędem, algorytm zmiennej metryki, RPROP [RB93], metoda gradientów sprzężonych, algorytm Levenberga-Marquardta [HM94], rekurencyjne najmniejsze kwadraty [BR98], AdaGrad [DHS11], RMSProp [HSS12], czy wreszcie najbardziej powszechny aktualnie aglortym Adam [KB14]. Część spośród wymienionych tu algorytmów omówiono dalej w punktach 4.3.3–4.3.9.

4.3.2 Wsteczna propagacja dla prostego przykładu sieci

Sieci neuronowe mogą tworzyć różne skomplikowane struktury. W tym miejscu postaramy się przybliżyć jedną z podstawowych i najprostszych struktur — sieć z jedną warstwą ukrytą i jednym wyjściem, która może zostać użyta do zadania aproksymacji danych. Schemat sieci przedstawia rys. 4.6. Sieć ta zawiera N neuronów w warstwie ukrytej. W każdym z neuronów obliczana jest najpierw odpowiedź bloku sumowania s_k:

$$s_k = v_{k0} + \sum_{j=1}^{n} v_{kj} x_j, \quad k = 1, \ldots, N. \tag{4.29}$$

Następnie obliczana jest odpowiedź bloku funkcji aktywacji:

$$\phi(s_k) = \frac{1}{1 + e^{-s_k}}, \quad k = 1, \ldots, N. \tag{4.30}$$

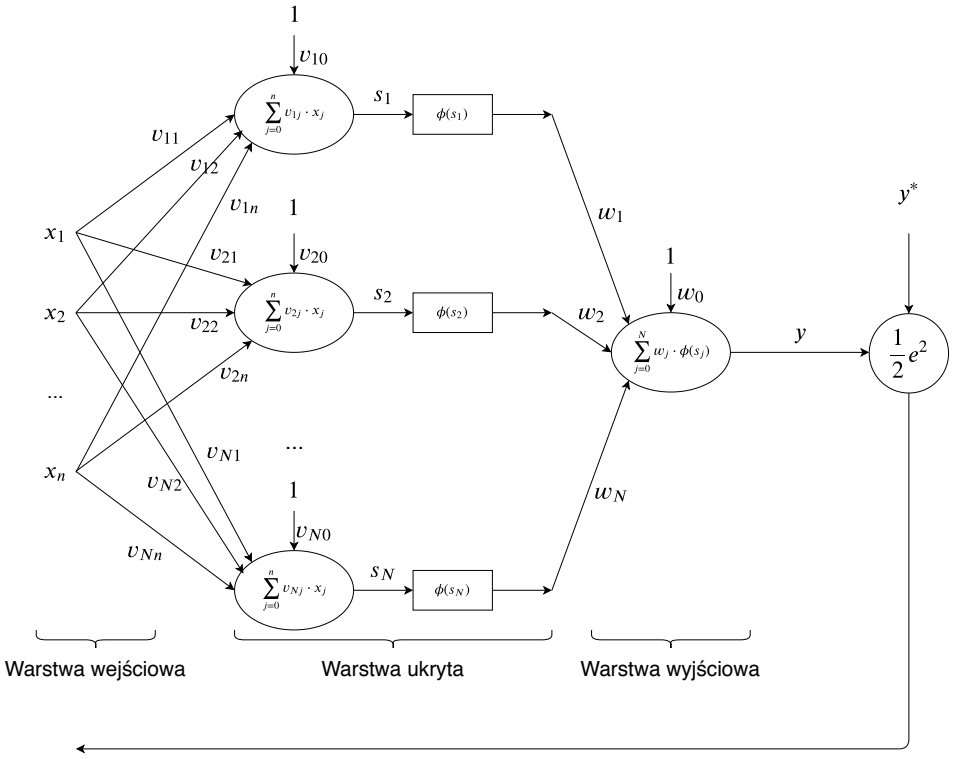
W powyższym przykładzie została użyta sigomoidalna unipolarna[6] funkcja aktywacji. Na wyjściu sieci znajduje się pojedynczy neuron z liniową funkcją aktywacji. Liniowa funkcja aktywacji może zostać pominięta, a działanie warstwy wyjściowej posiada następującą postać:

$$y = w_0 + \sum_{k=1}^{N} w_k \, \phi(s_k). \tag{4.31}$$

Wzór na uczenie sieci neuronowej bazuje na wzorze (4.26). Błąd sieci może zostać uproszczony do postaci:

$$\frac{1}{2} e^2 = \frac{1}{2} (y - y^*)^2, \tag{4.32}$$

[6] Przymiotnik unipolarna oznacza, że funkcja przyjmuje wartości z przedziału $(0,\ 1)$, funkcja bipolarna przyjmowałaby wartości z przedziału $(-1,\ 1)$.



Rys. 4.6: Szczegółowy schemat działania sieci neuronowej z jedną warstwą ukrytą (źródło: *opracowanie własne*).

gdzie y to odpowiedź sieci neuronowej, a y^* to wartość oczekiwana dla danych wejściowych. Znając ogólny wzór (4.26), można wyprowadzić z niego wzory poprawiające wagi w_k oraz wagi v_{kj}. W przypadku wag v_{kj} wyprowadzenie ma następującą postać:

$$\begin{aligned}\frac{\partial \frac{1}{2}e^2}{\partial v_{kj}} &= \frac{\partial \frac{1}{2}e^2}{\partial y}\cdot\frac{\partial y}{\partial \phi(s_k)}\cdot\frac{\partial \phi(s_k)}{\partial s_k}\cdot\frac{\partial s_k}{\partial v_{kj}}\\ &= \frac{\partial \frac{1}{2}(y-y^*)^2}{\partial y}\cdot\frac{\partial(w_0+w_1\phi(s_1)+\cdots+w_k\phi(s_k)+\cdots+w_N\phi(s_N))}{\partial \phi(s_k)}\cdot\\ &\quad\cdot\frac{\partial \phi(s_k)}{\partial s_k}\cdot\frac{\partial(v_{k0}+v_{k1}x_1+\cdots+v_{kj}x_j+\cdots+v_{kn}x_n)}{\partial v_{kj}}\\ &= (y-y^*)\cdot w_k\cdot\phi(s_k)\,(1-\phi(s_k))\cdot x_j.\end{aligned} \tag{4.33}$$

Warto wyjaśnić, że fragment związany z pochodną funkcji aktywacji postaci:

$$\frac{\partial \phi(s_k)}{\partial s_k} = \frac{e^{-s_k}}{(1+e^{-s_k})^2} = \frac{e^{-s_k}+1-1}{(1+e^{-s_k})^2} = \phi(s_k)(1-\phi(s_k)), \tag{4.34}$$

wynika z własności sigmoidalnej funkcji aktywacji. Pozwala on obliczyć pochodną na podstawie znajomości obliczonej wcześniej wartości funkcji w punkcie (czyli wykonanego już wcześniej przebiegu obliczeń sieci w przód).

Z kolei wyprowadzenie dla wag w_k ma postać:

$$\begin{aligned}\frac{\partial \frac{1}{2}e^2}{\partial w_k} &= \frac{\partial \frac{1}{2}e^2}{\partial y} \cdot \frac{\partial y}{\partial w_k} \\ &= \frac{\partial \frac{1}{2}(y-y^*)^2}{\partial y} \cdot \frac{\partial (w_0 + w_1\phi(s_1) + \cdots + w_k\phi(s_k) + \cdots + w_N\phi(s_N))}{\partial w_k} \\ &= (y-y^*)\cdot\phi(s_k).\end{aligned} \tag{4.35}$$

W efekcie otrzymujemy dwa wzory pozwalające na korygowanie wag sieci:

$$v_{kj} := v_{kj} - \eta \cdot (y-y^*) \cdot w_k \cdot \phi(s_k)(1-\phi(s_k)) \cdot x_j, \tag{4.36}$$

$$w_k := w_k - \eta \cdot (y-y^*) \cdot \phi(s_k). \tag{4.37}$$

Na rysunkach 4.7–4.9 przedstawiono przykład zastosowania sieci jednowarstwowej do aproksymacji pewnej funkcji dwóch zmiennych. Rys. 4.7 obrazuje powierzchnię funkcji, którą chcemy zaproksymować i zaczerpnięte z niej $m = 1000$ punktów jako przykładów uczących (wartości y_i z dodanym szumem normalnym $\sim N(0,1)$). Rysunki 4.8 i 4.9 obrazują uzyskane dla tych danych dwa przykładowe aproksymatory — sieci neuronowe odpowiednio o $N = 4$ neuronach (model mniej dokładny) i $N = 16$ neuronach (model bardziej dokładny).

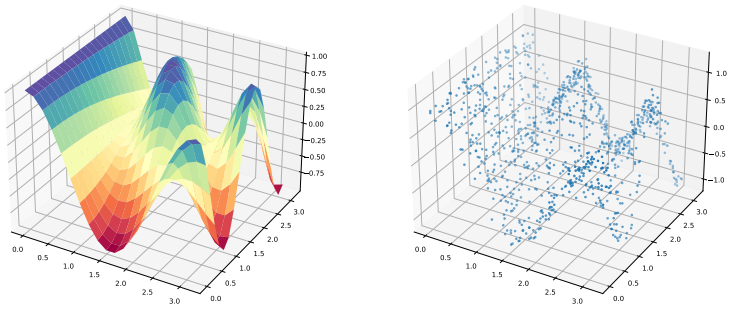
4.3.3 Uczenie z rozpędem — Momentum Backpropagation

Metoda ta polega na dodaniu do wzoru na korektę dowolnej wagi (parametru) sieci tzw. składnika momentum (lub inaczej rozpędu), który jest ułamkniem korekty z wcześniejszego kroku [Klę05]. Dla czytelności zapisów zaniedbajmy chwilowo indeksy wag, a także oznaczmy różnicę $w(t+1) - w(t)$ jako $\Delta w(t)$. Opisany powyżej zmodyfikowany wzór na korektę dowolnej wagi ma zatem postać:

$$\Delta w(t) = -\eta \frac{\partial \frac{1}{2}e^2(t)}{\partial w(t)} + \nu \Delta w(t-1), \tag{4.38}$$

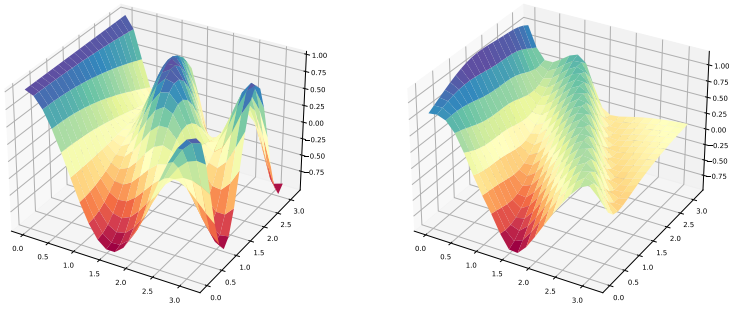
gdzie ν jest tzw. współczynnikiem momentum (ang. *momentum rate*) wybieranym z przedziału $[0,1)$. Zwykle przyjmuje się stosunkowo wysokie wartości ν, np. $\nu = 0.9$ [BŚ00; Oso00; RM99].

funkcja aproksymowana dane uczące ($m = 1000$, $\sim N(0, 0.1)$)



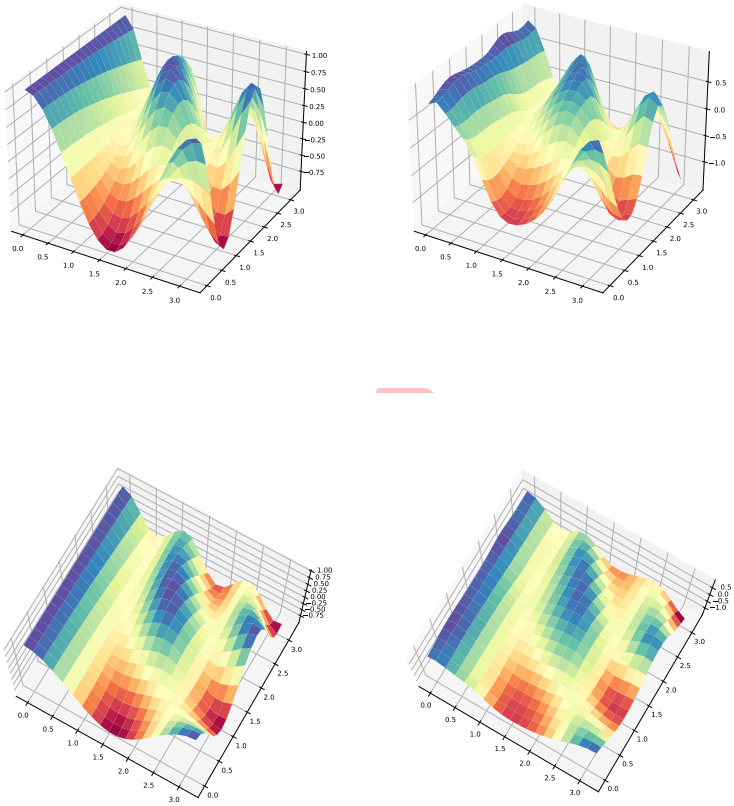
Rys. 4.7: Funkcja dwóch zmiennych i pobrane z niej zaszumione próbki uczące (źródło: *opracowanie własne*).

funkcja aproksymowana aproksymator neuronowy ($N = 4$)



Rys. 4.8: Funkcja aproksymowana i aproksymująca ją sieć neuronowa (1 warstwa ukryta neuronów z sigmoidalną funkcją aktywacji) otrzymana dla nastaw: $N = 4$, 10^5 iteracji uczących, $\eta = 0.005$, 32-elementowy wsad uczący. Średni błąd kwadratowy (MSE): 0.1013, Współczynnik dopasowania modelu (R^2): 0.6254 (źródło: *opracowanie własne*).

funkcja aproksymowana aproksymator neuronowy ($N = 16$)



Rys. 4.9: Funkcja aproksymowana i aproksymująca ją sieć neuronowa (1 warstwa ukryta neuronów z sigmoidalną funkcją aktywacji) otrzymana dla nastaw: $N = 16$, 10^6 iteracji uczących, $\eta = 0.005$, 32-elementowy wsad uczący. Średni błąd kwadratowy (MSE): 0.0212. Współczynnik dopasowania modelu (R^2): 0.9216 (źródło: *opracowanie własne*).

Dzięki składnikowi rozpędu zmiana wagi zależy nie tylko od samego gradientu w aktualnym punkcie, ale i od zmian tej wagi z wcześniejszych kroków, jako że zgodnie z regułą (4.45) wyraz $\Delta w(t-1)$ zagnieżdża w sobie rekurencyjnie wyrazy $\Delta w(t-2)$, $\Delta w(t-3)$, itd. Nadaje to procesowi uczenia pewnej bezwładności, która powoduje, że kierunek zmian wag jest z kroku na krok modyfikowany nieznacznie,

o ile nie przeciwstawi mu się całkowicie aktualny gradient [Klę05].

W skrócie można powiedzieć, że metoda momentum wygładza proces uczenia w stosunku do podstawowej metody gradientowej. Efekt wygładzenia wnosi dwie zalety:

1. niweluje oscylacje w sytuacji gdy proces uczenia natrafi na obszar powierzchni błędu przypominający „stromy wąwóz", który opada delikatnie wzdłuż swojej długości; podstawowa metoda gradientowa powodowałaby wówczas przeskakiwanie z jednego na drugi brzeg wąwozu, podczas gdy efektywne poruszanie się w kierunku opadania wąwozu byłoby powolne,
2. przyspiesza proces uczenia na powierzchniach błędu o małym nachyleniu, tj. wtedy gdy wartości $\partial \frac{1}{2}e^2(t)/\partial w(t)$ są bardzo małe.

Obie te zalety wynikają z faktu, że w metodzie momentum zgodne składowe kolejnych gradientów wzmacniają się, natomiast przeciwne wzajemnie znoszą [Klę05].

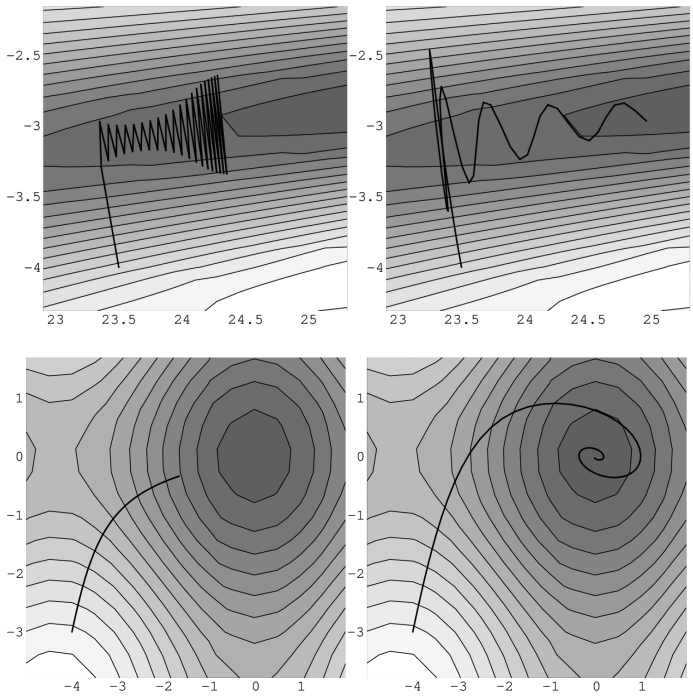
Rekurencyjne rozwinięcie wzoru (4.45) jest następujące:

$$\begin{aligned}
\Delta w(t) &= -\eta \frac{\partial \frac{1}{2}e^2(t)}{\partial w(t)} + \nu \Delta w(t-1) \\
&= -\eta \frac{\partial \frac{1}{2}e^2(t)}{\partial w(t)} + \nu \left(-\eta \frac{\partial \frac{1}{2}e^2(t-1)}{\partial w(t)} + \nu \Delta w(t-2) \right) \\
&= -\eta \frac{\partial (\frac{1}{2}e^2(t))}{\partial w(t)} + \nu \left(-\eta \frac{\partial \frac{1}{2}e^2(t-1)}{\partial w(t-1)} + \nu \left(-\eta \frac{\partial \frac{1}{2}e^2(t-2)}{\partial w(t-2)} + \cdots \right) \right) \\
&= -\eta \sum_{k=0}^{\infty} \nu^k \frac{\partial \frac{1}{2}e^2(t-k)}{\partial w(t-k)}.
\end{aligned} \tag{4.39}$$

Jak można zauważyć, wzór na korektę wagi jest sumą wykładniczą po wszystkich wcześniejszych wyrazach gradientu. Ponieważ $\nu < 1$, wkład dawnych gradientów maleje wykładniczo z każdym krokiem uczenia. Największy wpływ na sumę mają stosunkowo niedawne gradienty. Dla $\nu \to 0^+$ proces uczenia szybko „zapomina" o niedawnych gradientach i szybko reaguje według aktualnego gradientu. Dla $\nu \to 1^-$ proces uczenia „ma długą pamięć", jest stabilny, ale wolno reaguje na nowe gradienty [RM99].

Przykładowe ilustracje porównujące działanie zwykłej metody gradientowej z metodą momentum pokazano na rys. 4.10.

Wspomniane wcześniej przyspieszenie na płaskich obszarach powierzchni błędu można pokazać następująco. Przyjmując, że w rozwinięciu wykładniczym



Rys. 4.10: Porównanie działania zwykłej metody gradientowej (po lewej stronie) z metodą momentum (po prawej stronie) w pewnej przestrzeni dwóch wag. Rysunki pokazują stan uczenia po takiej samej liczbie kroków algorytmu (źródło: (Klę05)).

(4.39) kolejne gradienty są bardzo małe i stałe, $\frac{\partial \frac{1}{2}e^2(t)}{\partial w(t)} = G = \text{const.}$, otrzymujemy:

$$\begin{aligned}\Delta w(t) &= -\eta \sum_{k=0}^{\infty} \nu^k \frac{\partial \frac{1}{2}e^2(t-k)}{\partial w(t-k)} = -\eta G \sum_{k=0}^{\infty} \nu^k \\ &= -\frac{\eta}{1-\nu} G. \end{aligned} \tag{4.40}$$

Ostatnie przejście wykorzystuje fakt, że $\sum_{k=0}^{\infty} \nu^k = \lim_{n\to\infty} \frac{1-\nu^{n+1}}{1-\nu} = \frac{1}{1-\nu}$ dla $|\nu| < 1$. Traktując teraz wyrażenie $\eta/(1-\nu)$ jako efektywny współczynnik uczenia, zauważamy, że przewyższa on zwykły współczynnik uczenia η (lub jest mu równy), dzięki dzielnikowi $0 < 1-\nu \leqslant 1$. Zatem np. dla $\nu = 0.9$ uzyskujemy na płaskiej powierzchni błędu dziesięciokrotne przyspieszenie uczenia w stosunku do uczenia, w którym korekty uwzględniałyby tylko wyrażenie $-\eta G$.

Przy stosowaniu metody momentum należy pamiętać, że składnik rozpędu nie powinien całkowicie zdominować procesu uczenia, gdyż może to prowadzić do niestabilności algorytmu. Dobrze jest na bieżąco obserwować wartość funkcji błędu w procesie uczenia, dopuszczając do jej wzrostu w ograniczonym zakresie, np. o 5%. Jeżeli próg ten miałby zostać naruszony, to należy zignorować składnik rozpędu poprzez wymuszenie $\Delta w(t-1)=0$. Spowoduje to, że składnik gradientowy odzyska na nowo decydujący wpływ na proces uczenia [Oso00].

4.3.4 Resilient Backpropagation — RPROP

Metodę RPROP (ang. *Resilient backPROPagation*) zaproponowali w 1993 r. Riedmiller i Braun [RB93]. Jest ona przeznaczona dla pełnego lub inaczej wsadowego trybu korekcji wag sieci neuronowej — tryb *off-line* . Oznacza to, że pojedyncza poprawka następuje dopiero po „obejrzeniu" przez sieć całego zbioru uczącego i obliczeniu sumarycznego a tym samym dokładnego gradientu (a nie jak było to w trybie on-line — po każdym przykładzie uczącym).

Kluczowymi elementami podejścia RPROP są:

- wykorzystywanie jedynie samego znaku każdej składowej gradientu (natomiast wartości są pomijane),
- przypisanie indywidualnego (prywatnego) współczynnika uczenia do każdej wagi (parametru) sieci,
- modyfikowanie współczynników uczenia po każdym kroku.

Współczynnik uczenia danej wagi jest zwiększany, gdy znaki kolejnych gradientów pozostają zgodne, natomiast zmniejszany (a dokładnie połowiony), gdy są różne. Ten mechanizm rekompensuje fakt pomijania wartości (długości) gradientu. Należy zwrócić uwagę, że w większości innych metod uczenia dla sieci neuronowych współczynniki uczenia pozostają stałe.

Podobnie jak w poprzednim punkcie, niech $w(t)$ oznacza dowolną wagę sieci (z pominięciem indeksów dla czytelności). Główny wzór, za pomocą którego w metodzie RPROP poprawiane są wagi, ma postać:

$$w(t+1)=w(t)-\eta_w(t)\,\mathrm{sgn}\,\frac{\partial \frac{1}{2}E^2(t)}{\partial w(t)}. \tag{4.41}$$

We wzorze tym należy zwrócić uwagę na: zależność funkcyjną współczynnika uczenia od kroku czasowego (i fakt, że jest on skojarzony z konkretną wagą) — $\eta_w(t)$, oraz na funkcję błędu $\frac{1}{2}E^2(t)$, która oznacza sumaryczny błąd kwadratowy (suma po całym zbiorze uczącym), tj.:

$$\frac{1}{2}E^2(t)=\sum_{i=1}^{m}\frac{1}{2}e_i^2(t). \tag{4.42}$$

Można zatem powiedzieć, że $\partial \frac{1}{2}E^2(t)/\partial w(t)$ reprezentuje dokładną wartość gradientu wzdłuż składowej w.

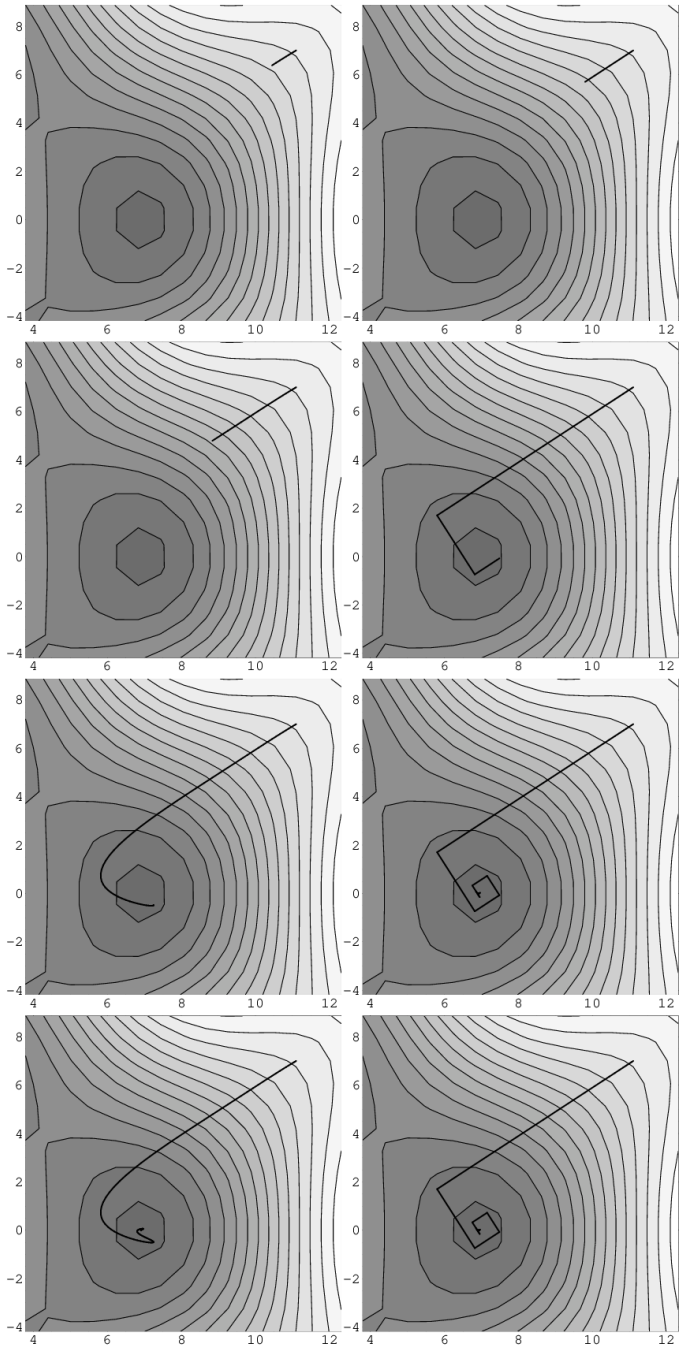
Bardzo istotnym elementem metody jest wzór na modyfikację każdego współczynnika uczenia, który ma postać:

$$\eta_w(t) = \begin{cases} \min\{a\,\eta_w(t-1), \eta_{\max}\}, & \text{dla } \frac{\partial \frac{1}{2}E^2(t-1)}{\partial w(t-1)} \cdot \frac{\partial \frac{1}{2}E^2(t)}{\partial w(t)} > 0; \\ \max\{b\,\eta_w(t-1), \eta_{\min}\}, & \text{dla } \frac{\partial \frac{1}{2}E^2(t-1)}{\partial w(t-1)} \cdot \frac{\partial \frac{1}{2}E^2(t)}{\partial w(t)} < 0; \\ \eta_w(t-1), & \text{w przeciwnym razie.} \end{cases}$$

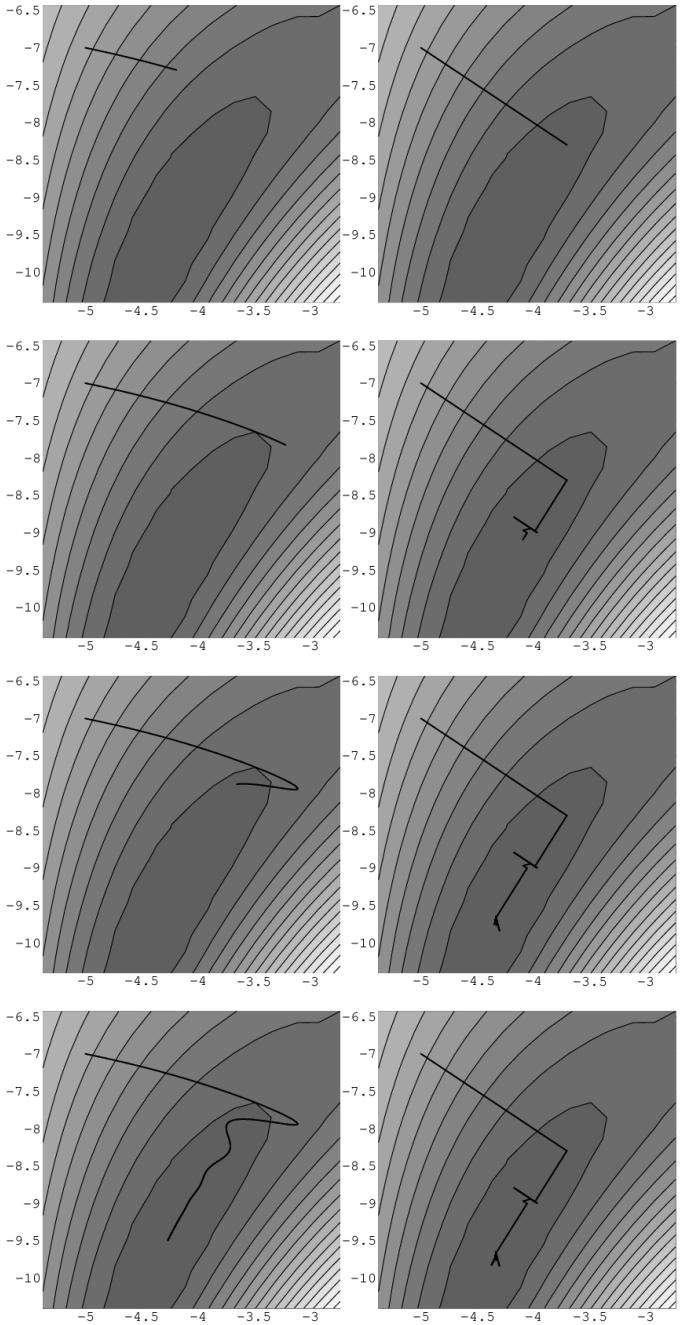
Zwyczajowo przyjmuje się następujące wartości dla stałych występujących w tym wzorze: $a = 1.2$ (tempo zwiększania współczynnika uczenia), $b = 0.5$ (tempo zmniejszania współczynnika uczenia), $\eta_{\min} = 10^{-6}$, $\eta_{\max} = 10^2$ (skrajne wartości współczynnika uczenia). W chwili startu można przyjąć stosunkowo małe wartości początkowe, np. $\eta_w(0) = 0.01$.

Rysunki 4.11 i 4.12 ilustrują porównanie działania uczenia gradientowego z rozpędem (metoda momentum) z metodą RPROP.

Należy zdawać sobie sprawę, że dokładność i szybkość zbieżności metody RPROP jest okupiona kosztem obliczenia sumarycznego błędu kwadratowego (jako że jest to metoda off-line). Dla zbiorów danych o odpowiednio dużej liczbie przykładów (np. dla m rzędu 10^6) obliczanie takiego błędu przed każdą poprawką może być zbyt kosztowne i niepraktyczne. W takich sytuacjach preferuje się uczenie on-line lub pewne podejścia mieszane (np. bootstrap) polegające na czerpaniu pewnego losowego podzbioru z próby uczącej w każdym kroku do obliczenia błędu.



Rys. 4.11: Porównanie działania metody momentum (po lewej stronie) z metodą RPROP (po prawej stronie) w pewnej przestrzeni dwóch wag (przykład pierwszy). Rysunki pokazują stan uczenia po takiej samej liczbie kroków algorytmu (źródło: (Klę05)).



Rys. 4.12: Porównanie działania metody momentum (po lewej stronie) z metodą RPROP (po prawej stronie) w pewnej przestrzeni dwóch wag (przykład drugi). Rysunki pokazują stan uczenia po takiej samej liczbie kroków algorytmu (źródło: (Klę05)).

4.3.5 Wykładnicze średnie kroczące i współczesne metody gradientowe dla sieci neuronowych

Dla uproszczenia i ujednoliczenia notacyjnego w poniższych sekcjach będziemy pisali tylko W dla oznaczenia wektora *wszystkich* wag w danej sieci oraz tylko ∇ dla oznaczenia gradientu czyli wektora wszystkich pochodnych cząstkowych funkcji błędu ze względu na poszczególne wagi. Nazwy czy też numery warstw nie będą wyróżniane. Natomiast, będziemy zapisywali indeksy dolne jako numery kroków czasowych (litera t), tj. kroków, w których mają miejsce aktualizacje wag, przyjmując że $t = 0$ stanowi chwilę początkową. Należy także wyjaśnić, że jeżeli proces uczenia jest realizowany w ramach pewnej liczby tzw. *epok*, a każda z nich składa się z kolei z pewnej liczby wsadów (ang. *mini-batches*) pokazywanych sieci, to kolejne chwile czasowe są tak naprawdę skojarzone właśnie ze wsadami (a nie z epokami), bo to po „obejrzeniu" pewnego wsadu następują aktualizacje wag. Na przykład, mając ustalone 10 wsadów przypadających na każdą epokę, krok z chwili $t = 35$ na chwilę $t = 36$ należy rozumieć jako szóstą aktualizację wektora wag sieci w trakcie trwania czwartej epoki.

Wyjaśnijmy najpierw sam matematyczny mechanizm wykładniczej średniej kroczącej (ang. EMA — *Exponenential Moving Averages*) w oderwaniu od kontekstu sieci neuronowych. Jest to mechanizm, który pozwala nam aktualizować na bieżąco pewną obserwowaną średnią wraz z napływającymi danymi. Dodatkowo, w zależności od wybranej wartości dla występującego w tym mechanizmie parametru β, obliczana średnia może mieć charakter bardziej długoterminowy lub krótkoterminowy.

Mając dany pewien ciąg liczb (lub wektorów) $x_0, x_1, \ldots$, rozważmy nowy ciąg zbudowany w sposób następujący:

$$y_t := \beta y_{t-1} + (1-\beta)x_t, \quad t = 0, 1, \ldots; \tag{4.43}$$

gdzie $y_{-1} = 0$ na potrzeby inicjalizacji, a parametr $\beta \in [0, 1)$ jest współczynnikiem wykładniczego wygaszania (ang. *decay rate*), często ustawianym przynajmniej na wartość 0.9 w typowych zastosowaniach. Rozwinięcie wzoru (4.43) generuje następujący wzorzec:

$$\begin{aligned} y_t &:= \beta^2 y_{t-2} + (1-\beta)(x_t + \beta x_{t-1}) \\ &= \beta^3 y_{t-3} + (1-\beta)(x_t + \beta x_{t-1} + \beta^2 x_{t-2}) \\ &= \cdots = \beta^{t+1} \underbrace{y_{-1}}_{0} + (1-\beta) \sum_{i=0}^{t} \beta^{t-i} x_i, \end{aligned} \tag{4.44}$$

gdzie występujące pod sumą współczynniki β^{t-i} zmieniają się zgodnie z postępem geometrycznym. Wartość β bliska jedynce powoduje efekt tzw. „długiej

pamięci", ponieważ wiele przeszłych wyrazów x_i wnosi wkład w obliczane y_t. Taki efekt można nazwać wykładniczym wygładzaniem długookresowym. Z kolei w przypadku przeciwnym, nastawienie wartości β bliskiej zeru, przekłada się na tzw. „krótką pamięć", ponieważ współczynniki β^{t-i} sięgające dalego w przeszłość są bardzo szybko wygaszane. Ponadto, winno się zauważyć, że wzór (4.44) stanowi de facto ważoną średnią *obciążoną* (nieunormowaną), ponieważ współczynniki wagujące nie sumują się do jedności, z wyjątkiem przypadku $\beta = 0$, który implikuje $y_t = x_t$ dla wszystkich t. Mówiąc dokładniej, dla ustalonego t suma współczynników wagujących wynosi $(1-\beta^{t+1})$. A odwrotność tej liczby może stanowić stałą normalizującą, która bywa stosowana w niektórych podejściach (m.in. w algorytmie Adam) w celu zdjęcia wspomnianego obciążenia średniej, co bywa istotne szczególnie dla chwil początkowych.

4.3.6 Uczenie z rozpędem — podejście tradycyjne a podejście współczesne

Tak jak omówiono to już w punkcie 4.3.3, technika uczenia z rozpędem [Pol64; RHW86] nakazuje name podczas obliczania poprawki wag dołożyć do aktualnego gradientu pewien ułamek poprzedniej poprawki. Zgodnie z przyjętą w tym punkcie notacją, tradycyjne uczenie z rozpędem można zapisać w poniższy sposób:

$$W_{t+1} := W_t - \eta \nabla_t + \nu(W_t - W_{t-1}). \tag{4.45}$$

Poprawna inicjalizacja wymaga, aby dla chwili $t = 0$ przyjąć wektor zerowy jako sztuczną (nieistniejącą) poprzednią poprawkę, tzn. $W_0 - W_{-1} = \mathbf{0}$.

Dla przypomnienia — rekurencyjne rozwinięcie wzoru (4.45) pokazuje nam, że jest on średnią ważoną wszystkich przeszłych gradientów ze współczynnikami wagującymi gasnącymi wykładniczo, tzn.:

$$\begin{aligned} W_{t+1} &:= W_t - \eta\nabla_t - \nu\,\eta\nabla_{t-1} + \nu^2(W_{t-1} - W_{t-2}) \\ &= \cdots = W_t - \eta \sum_{i=0}^{t} \nu^{t-i}\nabla_i. \end{aligned} \tag{4.46}$$

Pomimo przyspieszenia w procesie uczenia, które to podejście może wnieść (szczególnie dla płaskich regionów funkcji błędu), warto uczciwie zaznaczyć, że w niektórych sytuacjach zbyt duży rozpęd może zagrozić zbieżności i stabilności algorytmu. Dlatego też tradycyjne implementacje bywają wyposażane w „bezpiecznik", który resetuje rozpęd (tzn. wymusza ostatni składnik we wzorze (4.45) jako zerowy), jeżeli obserwowana wartość błędu wzrośnie o pewien określony procent zamiast zmaleć.

Bardziej współczesna wersja uczenia z rozpędem jest oparta właśnie o wykładniczą średnią kroczącą i jest powszechnie formułowana następująco:

$$m_t := \beta m_{t-1} + (1-\beta)\nabla_t; \tag{4.47}$$
$$W_{t+1} := W_t - \eta\, m_t; \tag{4.48}$$

gdzie $m_{-1} = \mathbf{0}$ stanowi wyraz inicjalizujący. Po rozwinięciu rekurencyjnym zgodnym ze wzorem (4.44) otrzymujemy

$$W_{t+1} := W_t - \eta\,(1-\beta)\sum_{i=0}^{t}\beta^{t-i}\nabla_i. \tag{4.49}$$

Oczywiście, występujący powyżej współczynnik β jest odpowiednikiem ν z tradycyjnej wersji uczenia z rozpędem. Jednakże, należy zauważyć ważną różnicę pomiędzy wzorami (4.46) i (4.49), mianowicie — obecność mnożnika $1-\beta$, który pojawia się przed sumą we wzorze (4.49). Mnożnik tem przeciwdziała bardzo dużym rozpędom. Jeżeli β zmierzałaby granicznie ku wartości 1, to współczynniki wagujące występujące w rozwinięciu nie malałyby szybko i tym samym skumulowany rozpęd sięgałby daleko w przeszłość, ale równocześnie wtedy mnożnik $1-\beta$ staje się bliski zeru i ogranicza efekt takiego nadmiernego rozpędu.

4.3.7 AdaGrad i RMSProp

Algorytm AdaGrad (ang. *Adaptive Gradient*) [DHS11] adaptuje efektywny współczynnik uczenia poprzez wprowadzenie podzielnika przypominającego odchylenie standardowe. Dokładniej mówiąc, podzielnik ten jest pierwiastkiem ze skumulowanych przeszłych gradientów (włączając także ten najświeższy). Wzory potrzebne do aktualizacji wag sieci neuronowej przyjmują następującą postać:

$$v_t := v_{t-1} + \nabla_t^2, \tag{4.50}$$
$$W_{t+1} := W_t - \frac{\eta}{\sqrt{v_t+\varepsilon}}\nabla_t, \tag{4.51}$$

gdzie $v_{-1} = \mathbf{0}$, $\varepsilon = 10^{-7}$, $\nabla_t^2 = \nabla_t \circ \nabla_t$. Stała ε jest wentylem bezpieczeństwa zapobiegającym dzieleniu przez zero[7]. W powyższych wzorach zarówno operacja dzielenia jak i $\sqrt{\cdot}$ są zdefiniowane pokropkowo, tzn. na rzecz poszczególnych elementów wektora, i w związku z tym efektywne współczynniki uczenia dla poszczególnych składowych gradientu różnią się, stają się indywidualne (ale w sposób niejawny). Warto tę uwagę zestawić z omówioną wcześniej starszą metodą RPROP,

[7] Dla przykładu biblioteka Keras przeznaczona dla głębokich sieci neuronowych, nazywa stałą ε mianem *fuzz factor*.

gdzie każda waga sieci miała swój prywatny współczynnik uczenia zdefiniowany i przechowywany jawnie.

Drugi algorytm — RMSProp [HSS12] — który przedstawiamy w tym punkcie, został zaproponowany w 2012 r. przez Hintona i in. Algorytm ten został pomyślany jako pewnego rodzaju wariant podejścia RPROP, który byłby odpowiedni dla uczenia mini-wsadami, a nie tylko trybu off-line (jak to jest sugerowane w oryginalnym RPROP). Matematycznie, algorytm RMSProp można rozumieć jako algorytm AdaGrad wyposażony w mechanizm wykładniczej średniej kroczącej. Wzory aktualizacyjne przyjmują następującą postać:

$$v_t := \beta v_{t-1} + (1-\beta)\nabla_t^2, \tag{4.52}$$

$$W_{t+1} := W_t - \frac{\eta}{\sqrt{v_t + \varepsilon}}\nabla_t, \tag{4.53}$$

gdzie $v_{-1} = \mathbf{0}$, a $\beta = 0.9$ jest popularnym wyborem (ponownie).

4.3.8 Adam

Niniejszy punkt poświęcony jest szczególnie ważnemu algorytmowi ***Adam***[8], który został zaproponowany w 2014 r. przez Kingmę i Ba [KB14]. Algorytm ten po dziś dzień jest uważany za tzw. state-of-the-art w ramach optymalizatorów sieci neuronowych i w powszechnie używanych bibliotekach (Keras, TensorFlow, Torch itp.) stanowi on algorytm domyślny. Na ten fakt złożyły się zarówno prostota implementacji jak i skuteczność Adama obserwowana dla wielu zbiorów danych.

Z matematycznego punktu widzenia algorytm Adam stanowi swoiste połączenie uczenia z rozpędem i algorytmu RMSProp. Adam używa estymat momentów gradientu (momenty rozumiane statystycznie) — momentu pierwszego i momentu drugiego niecentralnego (lub inaczej: niecentralnej wariancji). Estymaty te są obliczane poprzez wykładnicze średnie kroczące, przy czym autorzy propozycji zadbali o zdjęcie obciążenia, które ma miejsce w takiej średniej. Obliczone estymaty pozwalają na adaptację długości poprawki, co powoduje że tzw. efektywny współczynnik uczenia staje się w sposób niejawny indywidualny dla każdej wagi (podobnie jak ma to miejsce w RMSProp).

Mówiąc dokładniej, algorytm śledzi dwie wykładnicze średnie kroczące: średnią gradientów (ciąg m_t) oraz kwadratów gradientów (ciąg v_t), stosując dwa parametry $\beta_1, \beta_2 \in [0,1)$ kontrolujące tempo wygaszania średnich. Co ciekawe, wartości tych parametrów zaproponowane w oryginalnej publikacji [KB14], tj. $\beta_1 0.9$ oraz $\beta_2 = 0.999$, są także po dziś dzień powszechnie stosowane jako domyślne, i dobrze sprawdzają się dla wielu problemów uczących.

[8] nazwa luźno nawiązująca do wyrażenia: *adaptive movement*

Rozpoczynając o wektorów zerowych tj. $m_{-1} = v_{-1} = \mathbf{0}$, kolejne estymaty pierwszego i drugiego momentu są obliczano jako:

$$m_t := \beta_1 m_{t-1} + (1-\beta_1)\nabla_t, \tag{4.54}$$

$$v_t := \beta_2 v_{t-1} + (1-\beta_2)\nabla_t^2, \tag{4.55}$$

a odpowiadające im rozwinięcia rekurencyjne są wynoszą:

$$m_t := (1-\beta_1)\sum_{i=0}^{t} \beta_1^{t-i}\nabla_i, \tag{4.56}$$

$$v_t := (1-\beta_2)\sum_{i=0}^{t} \beta_2^{t-i}\nabla_i^2. \tag{4.57}$$

Oczywiście, wzory (4.56) i (4.57) nie mają miejsca w faktycznej implementacji algorytmu, a podajemy je tu dla przypomnienia sensu działania wykładniczych średnich kroczących. Po odświeżeniu estymat (4.54), (4.55) dla aktualnego t, obliczane są ich skorygowane wersje $\widehat{m}_t$ oraz $\widehat{v}_t$, powzstałe poprzez zdjęcie obciążenia występującego w średnich kroczących za pomocą odpowiedniego podzielnika, tj.:

$$\widehat{m}_t := m_t/(1-\beta_1^{t+1}). \tag{4.58}$$

$$\widehat{v}_t := v_t/(1-\beta_2^{t+1}). \tag{4.59}$$

I wreszcie, aktualizacja wektora wag sieci neuronowej odbywa się z użyciem natępującego wzoru:

$$W_{t+1} := W_t - \eta\,\widehat{m}_t/(\sqrt{\widehat{v}_t}+\varepsilon), \tag{4.60}$$

gdzie $\varepsilon = 10^{-7}$.

Własności algorytmu Adam

Aby faktycznie przekonać się, czy średnie kroczące obliczane wg wzorów (4.54), (4.55) stanowią obciążone czy też nieobciążone estymatory, należy wyznaczyć ich wartości oczekiwane i sprawdzić matematycznie, czy spełnione są następujące równości: $\mathbb{E}(m_t) = \mathbb{E}(\nabla_t)$, $\mathbb{E}(v_t) = \mathbb{E}(\nabla_t^2)$. Poniżej przedstawiamy takie sprawdzenie (powtarzane za pracą [KB14]) tylko dla pierwszego momentu, ponieważ dla drugiego momentu jest ono analogiczne.

Wartość oczekiwana średniej kroczącej m_t może być obliczona w następujący sposób

$$\begin{aligned}\mathbb{E}(m_t) = \mathbb{E}\left((1-\beta_1)\sum_{i=0}^{t}\beta_1^{t-i}\nabla_i\right) &= (1-\beta_1)\sum_{i=0}^{t}\beta_1^{t-i}\mathbb{E}(\nabla_i) \\ &= (1-\beta_1)\sum_{i=0}^{t}\beta_1^{t-i}\mathbb{E}(\nabla_t) + \zeta = (1-\beta_1^{t+1})\mathbb{E}(\nabla_t) + \zeta, \end{aligned} \tag{4.61}$$

gdzie wprowadzone wyrażenie ζ reprezentuje błąd, pozwalający na zastąpienie wartości oczekiwanej $\mathbb{E}(\nabla_i)$ (dla dowolnego i) poprzez jego przybliżenie równe $\mathbb{E}(\nabla_t)$. Jak wyjaśniają Kingma i Ba [KB14], mamy do czynienia albo z $\zeta = \mathbf{0}$ wtedy gdy prawdziwe wartości oczekiwane $\mathbb{E}(\nabla_i)$ są stacjonarne, albo z ζ bardzo małym ze względu na współczynnik wygaszania β_1 wybrany w taki sposób, aby średnia wykładnicza przypisywała bardzo małe współczynniki gradientom z dalekiej przeszłości. Jak widać, oryginalne estymatory są *obciązone*, ponieważ $\mathbb{E}(m_t) \approx (1-\beta_1^{t+1})\mathbb{E}(\nabla_t) \neq \mathbb{E}(\nabla_t)$ i $\mathbb{E}(v_t) \approx (1-\beta_2^{t+1})\mathbb{E}(\nabla_t^2) \neq \mathbb{E}(\nabla_t^2)$, jednakże istniejące niewielkie obciążenie może być łatwo poprawione zgodnie ze wzorami (4.58) i (4.59).

Dodatkowo warto zaakcentować dwie poniższe własności algorytmu Adam. Po pierwsze efektywne długości poprawek obliczane podczas gradientowego spadku w ramach Adama są w przybliżeniu ograniczone poprzez zadaną stałą η, jako że w typowych statystycznie sytuacjach mamy $\widehat{m}_t/(\sqrt{\widehat{v}_t}+\varepsilon) \approx \pm 1$. Po drugie, Adam jest *niezmienniczy względem skali* gradientów. Dla przykładu, przeskalowanie wszystkich wyrazów ∇_i przez pewien mnożnik $c > 0$ powoduje równoważnie przeskalowanie ciągu $\widehat{m}_t$ również poprzez mnożnik c oraz przeskalowanie ciągu $\widehat{v}_t$ poprzez mnożnik c^2. Można zauważyć, że te współczynniki skalujące niwelują się w ramach wzoru na finalną aktualizację wektor wag sieci neuronowej, ponieważ zachodzi równość $\widehat{m}_t/\sqrt{\widehat{v}_t} = (c\,\widehat{m}_t)/\sqrt{c^2\,\widehat{v}_t}$, jeżeli na chwilę zaniedbamy matematycznie bardzo małe ε obecne we wzorze (4.60).

4.3.9 Inne pomysły: Nadam, Adamax, AMSGrad

Dzięki idei Nesterova [Nes83] niektóre techniki optymalizacyjne mogą zostać wyposażone w pewien mechanizm „przewidywania" polegający na patrzeniu jeden krok do przodu i obliczeaniu gradientu nie dla aktualnego pozycji określonej przez wektor W_t, ale dla przybliżonej przyszłej pozycji. Na przykład, wprowadzenie tego pomysłu do techniki uczenia z rozpędem jest znane pod nazwą NAG (Nesterov Accelerated Gradient) i może być zapisane w sposób następujący:

$$m_t := \beta m_{t-1} + (1-\beta)\nabla_{|W_t - \eta m_{t-1}}, \tag{4.62}$$

$$W_{t+1} := W_t - \eta\, m_t. \tag{4.63}$$

Dolny indeks postaci $W_t - \eta m_{t-1}$ przy ∇ wskazuje, że używamy poprzedniej średniej „prędkości" m_{t-1}, aby ustawić się w hipotetycznej pozycji $W_t - \eta m_{t-1}$ i traktujemy ją jako argument do obliczenia gradientu w punkcie, zamiast używać W_t jako argumentu. W ogólności tego typu przewidujące podejścia prowadzą do poprawy zbieżności [HSS12; Nes83; Rud16].

Opierając się na powyższym pomyśle, w 2016 r. Dozat zaproponował algorytm o nazwie *Nadam* (Adam + Nesterov) [Doz16]. Aby przedstawić ten algorytm

prosto, warto zauważyć, że krok aktualizacyjny zwykłego algorytmu Adam można równoważnie zapisać jako

$$W_{t+1} := W_t - \frac{\eta}{\sqrt{\widehat{v}_t}+\varepsilon}\left(\beta_1\widehat{m}_{t-1} + \frac{1-\beta_1}{1-\beta_1^{t+1}}\nabla_t\right). \tag{4.64}$$

Wzór ten otrzymujemy, podstawiając (4.54) do wzorów (4.58) i (4.60). Zachowując wszystkie pozostałe formuły, Nadam powstaje poprzez zastąpienie we wzorze (4.64) momentu poprzedniego $\widehat{m}_{t-1}$ za pomocą aktualnego $\widehat{m}_t$.

Ciekawym innym pomysłem jest także wariant znany pod nazwą *Adamax*, zaproponowany przez samych autorów Adam [KB14]. Zamiast używać odwrotności normy ℓ_2 przeszłych gradientów w celu skalowania bieżącego gradientu, autorzy spojrzeli ogólnie na normę ℓ_p. Zaobserwowali, że dla szczególnego przypadku $p \to \infty$ otrzymuje się zdumiewająco prosty i stabilny algorytm (co może nie mieć miejsca dla dużych ale skończonych wartości p). Aby wyprowadzić algorytm Adamax, Kingma i Ba definiują najpierw uogólnioną średnią kroczącą dla v_t jako:

$$v_t = \beta_2^p v_{t-1} + (1-\beta_2^p)|\nabla_t|^p = (1-\beta_2^p)\sum_{i=0}^{t}\beta_2^{p(t-i)}|\nabla_i|^p, \tag{4.65}$$

gdzie hiperparametr celowo zapisano jako β_2^p, tzn. z jawnie wyróżnioną p-tą potęgą (chwilowo traktujemy p jako ustalone)[9] Następnie, autorzy zdefiniowali wielkość $u_t = \lim_{p\to\infty} v_t^{1/p}$ i zauważyli, że

$$\begin{aligned} u_t &= \lim_{p\to\infty}(1-\beta_2^p)^{1/p}\left(\sum_{i=0}^{t}\beta_2^{p(t-i)}|\nabla_i|^p\right)^{1/p} \\ &= \max\left\{\beta_2^t|\nabla_0|, \beta_2^{t-1}|\nabla_1|, \ldots, \beta_2|\nabla_{t-1}|, |\nabla_t|\right\}, \end{aligned} \tag{4.66}$$

co odpowiada prostej i eleganckiej regule rekurencyjnej postaci: $u_t := \max\{\beta_2 u_{t-1}, |\nabla_t|\}$. Przypominamy, że operacja $\max\{\cdots\}$ jest również w tym kontekście wykonywana pokropkowo. Ostatecznie, cały algorytm Adamax można zapisać w poniższy sposób:

$$m_t := \beta_1 m_{t-1} + (1-\beta_1)\nabla_t, \tag{4.67}$$

$$u_t := \max\{\beta_2 u_{t-1}, |\nabla_t|\}, \tag{4.68}$$

$$W_{t+1} := W_t - \eta\, m_t \Big/ \left((1-\beta_1^{t+1})u_t\right), \tag{4.69}$$

[9] co jest równoważne pewnej innej ułamkowej wartości tego parametru, tzn.: $\beta_2' \in [0,1)$, gdzie $\beta_2' = \beta_2^p$

gdzie $m_{-1}=u_{-1}=\mathbf{0}$. Warto także zwrócić uwagę na brak korygowania obciążania w wielkości u_t.

Jako ostatni przykład warto przywołać algorytm *AMSGrad* — stosunkowo nowy wariant Adama opracowany przez Reddi'ego [Red+18] w 2018 r. AMSGrad utrzymuje nadal normę ℓ_2, ale w sposób wymuszony zapewnia, że aktualny drugi moment $\widehat{v}_t$ nigdy nie jest mniejszy od poprzedniego. Zachowując wszystkie istotne wzory, kluczowa różnica występująca w AMSGrad sprowadza się do dodatkowego kroku: $\widehat{v}_t := \max\{\widehat{v}_{t-1}, v_t\}$.

4.3.10 Inicjalizacja wag

Bardzo ważnym problemem poruszanym w tematyce sieci neuronowych jest inicjalizacja wag — czyli nadanie wagom pewnych losowych wartości początkowych. Dla osób rozpoczynających swoją przygodę z sieciami neuronowymi zagadenie to może wydawać się mało istotne na pierwszy rzut oka. Co ciekawe, stanowiło ono przeszkodę w rozwoju głębokich sieci neuronowych przez wiele lat, i zostało prawidłowo rozwiązane stosunkowo niedawno (w drugiej dekadzie aktualnego stulecia).

Problemy wynikające z niestarannej inicjalizacji wag znane są pod nazwami *zanikających* lub *wybuchających gradientów* (ang. *vanishing* or *exploding gradients*). Mówiąc bardzo powierzchownie, pierwszoplanową przyczyną pojawiania się ich jest głębokość sieci — im więcej warstw tym łatwiej o te problemy ze względu na wykładniczy wzrost lub spadek wariancji agregujących się poprzez warstwy sygnałów. Natomiast drugoplanową rolę grają też tutaj wybrane funkcje aktywacji. Na przykład w przypadku funkcji sigmoidalnej oraz inicjalizacji wag zbyt dużymi wartościami występuje zjawisko zwane *nasycaniem* się sigmoid. Polega ono na tym, że zbyt duże wartości wag powodują, że funkcja sigmoidalna przybiera postać bliską funkcji schodkowej i tym samym posiada pochodną bardzo bliską zeru. W konsekwencji wyznaczane poprawki są również bardzo bliskie zeru i proces uczenia jest mocno spowolniony. W ogólności, dla głębokich sieci składających się z kilkunastu, kilkudziesięciu warstw można łatwo sprawdzić, że prowadzone obliczenia czy to w przód czy w tył sieci mogą nie spełniać swojej zamierzonej roli, jeżeli nie dysponujemy dobrymi regułami inicjalizującymi wagi.

Do takich najbardziej znanych reguł zaliczamy obecnie *inicjalizację Glorot* [GB10] z roku 2010 oraz *inicjalizację He* z roku 2015. Obydwie reguły zostały odkryte na podstawie rekurencyjnej analizy wariancji sygnałów dla obliczeń w obu kierunkach (powtórzenie tej analizy wykracza poza zakres tego podręcznika), przy czym druga reguła jest uzupełnieniem pierwszej dedykowanym tylko dla funkcji aktywacji ReLU. Jednocześnie w obu przypadkach możemy zdecydować się na inicjalizację zgodną z rozkładem normalnym lub jednostajnym, przy czym kluczową

rolę odgrywa określenie zakresu poprzez, odpowiednio: odchylenie standardowe i szerokość przedziału.

Dla uproszczenia indeksacji w poniższych zapisach w_l symbolizuje dowolną wagę w l-tej warstwie sieci, zaś symboliczne stałe $\overrightarrow{N_l}$, $\overleftarrow{N_l}$ oznaczają liczbę sygnałów wchodzących do neuronów w tej warstwie, odpowiednio podczas obliczeń w przód oraz w tył[10].

Inicjalizacja Glorot (rozkład normalny):

$$w_i \sim N\left(0, \sqrt{2/(\overrightarrow{N_l} + \overleftarrow{N_l})}\right). \tag{4.70}$$

Inicjalizacja Glorot (rozkład jednostajny):

$$w_i \sim U\left(-\sqrt{6/(\overrightarrow{N_l} + \overleftarrow{N_l})}, \sqrt{6/(\overrightarrow{N_l} + \overleftarrow{N_l})}\right). \tag{4.71}$$

Inicjalizacja He (rozkład normalny):

$$w_i \sim N\left(0, \sqrt{2/\overrightarrow{N_l}}\right). \tag{4.72}$$

Inicjalizacja He (rozkład jednostajny):

$$w_i \sim U\left(-\sqrt{6/\overrightarrow{N_l}}, \sqrt{6/\overrightarrow{N_l}}\right). \tag{4.73}$$

[10] Dla omawianych wcześniej wielowarstwowych sieci perceptronowych stałe te odpowiadają po prostu liczbie neuronów w warstwach l i $l+1$ czyli oznaczeniom N_l, N_{l+1}.

4.4 Ćwiczenia laboratoryjne (MATLAB)

Ćwiczenie 4.1 **Napisz skrypt realizujący algorytm uczenia perceptronu prostego dla liniowo separowalnego zbioru danych na płaszczyźnie** Wygeneruj zbiór danych w sposób sztuczny, kontrolując jego rozmiar m, a także margines odstępu pomiędzy klasami. Dane przechowaj w formie macierzy o wymiarach $m \times 4$, gdzie kolejne kolumny przechowują wartości: $1, x_{i1}, x_{i2}, y_i$. Przedstaw dane w formie wykresu (polecenie `scatter`). Zaimplementuj algorytm uczący jako funkcję przyjmująca jako argumenty zbiór danych i współczynnik uczenia η. Uwaga: funkcja powinna być ogólna, tzn. pracować dla danych dowolnej wymiarowości. Jako rezultaty zwróć otrzymany wektor wag oraz liczbę wykonanych kroków aktualizacyjnych (licznik k). Sprawdź wpływ następujących zmian na licznik k: zmiana liczby przykładów (parametr m), zmiana współczynnika uczenia (parametr η), zmiana marginesu odstępu pomiędzy klasami.

Ćwiczenie 4.2 **Napisz skrypt realizujący algorytm uczenia perceptronu prostego z wykorzystaniem „podnoszenia wymiarowości”** Wygeneruj na płaszyźnie nieseparowalny liniwo zbiór danych określony nad prostokątem: $[0, 2\pi] \times [-1, 1]$. Punkty danych przebywające wewnątrz pętli $x_2 = \pm \sin x_1$ zalicz do klasy pozytywnej, a na zewnątrz do klasy negatywnej. Przedstaw dane w formie wykresu (polecenie `scatter`). Wprowadź parametr decydujący o żądanej wymiarowości docelowej przestrzeni cech. Podnieś wymiarowość danych zgodnie z przekształceniem Gaussa. Wykonaj uczenie perceptronem prostym, wprowadzając rozszerzony warunek stopu (maksymalna liczba kroków w sytuacji, gdy nie następuje tradycyjne zatrzymanie). Wykreśl na płaszczyźnie otrzymaną nieliniową granicę decyzyjną, korzystając z funkcji `contour` lub `contourf`. Sprawdź wpływ zadanej wymiarowości oraz parametru rządzącego szerokością dzwonów Gaussa na kształt granicy decyzyjnej.

Ćwiczenie 4.3 **Napisz skrypt realizujący działanie i uczenie sieci neuronowej typu perceptron wielowarstwowy (Multi-Layer Perceptron).** Polecenia do wykonania:

- Napisz skrypt realizujący działanie i uczenie sieci neuronowej. Skrypt powinien przyjmować na wejście następujące argumenty: zbiór danych, zadaną liczbę neuronów, zadaną liczbę kroków uczenia, współczynnik uczenia. Skrypt powinien zwracać na wyjściu macierz z nauczonymi wartościami wag V i wektor nauczonych wag W.
- Napisz skrypt rysujący (`surf`) wykres powierzchni sieci neuronowej reprezentowanej przez wagi V, W jako funkcji x_1, x_2. Ustal zakres osi odpowiadający zakresom funkcji aproksymowanej.
- Napisz skrypt generujący zbiór danych (zbiór próbek) pochodzących z funkcji dwóch zmiennych $y(x_1, x_2) = \cos(x_1 \cdot x_2) \cdot \cos(2 \cdot x_1)$ zdefiniowanej na dziedzinie: x_1, x_2 należącej do przedziału $[0, \pi]$. Przyjmij rozmiar zbioru danych $m = 1000$. Zbiór danych przechowuj w macierzy o wymiarach $mx3$, gdzie kolejne kolumny będą odpowiadały zmiennym x_1, x_2, y.

- Za pomocą funkcji MATLABa `scatter3` i `surf` sporządź wykresy odpowiednio zbioru próbek i funkcji aproksymowanej.
- Przeprowadź uczenie i zbierz wyniki. Sugerowane ustawienia (rzędy wielkości): liczba kroków uczenia $T \sim \{10^5, ..., 10^6\}$, liczba neuronów $N \sim \{10, 20, ..., 100\}$, współczynnik uczenia $\eta \sim \{10^{-3}, ..., 10^{-1}\}$. Początkowe wartości wylosowanych macierzy V, W powinny być bardzo małe $\sim [-10^{-3}, 10^{-3}]$ (lub jeszcze mniejszy rząd wielkości).
- Wyświetl oba wykresy powierzchni: funkcji aproksymowanej i sieci neuronowej (funkcji aproksymującej) oraz porównaj podobieństwo wizualnie np. nakładając oba wykresy na siebie.

Ćwiczenie 4.4 Przebadaj działanie sieci dla różnej liczby neuronów w warstwie ukrytej (wykorzystaj program z Ćwiczenia 4.3). Polecenia do wykonania:

- Zaczerpnij z aproksymowanej funkcji nowy zbiór uczący o rozmiarze $m = 200$, przy czym wartości y należy obciążyć pewnym losowym błędem, tj. $y = y(x_1, x_2) + \varepsilon$, gdzie $\varepsilon \sim N(0, 0.2)$ - błąd losowy o rozkładzie normalnym, średniej zero i odchyleniu standardowym 0.2. W MATLABie jest funkcja `randn()` losująca liczbę (lub macierz liczb) z rozkładu normalnego.
- Podziel losowo powyższy zbiór na część uczącą i część testową w proporcji 70:30.
- W pętli (wielokrotnie) przeprowadź proces uczenia sieci zadając coraz większą liczbę neuronów: $N = 10, 20, ..., 100$ (10 iteracji). Sieć ma być uczona tylko na zbiorze uczącym. Za każdym razem początkowe wartości wag V i W mają być wylosowane na nowo. W każdej z 10 iteracji po nauczeniu sieci należy obliczyć błąd popełniany przez nią na zbiorze uczącym i na zbiorze testowym (niewidzianym podczas uczenia) jako średnią różnicę bezwzględną pomiędzy oczekiwanymi wartościami y, a odpowiedziami sieci neuronowej. Obie wartości zapamiętaj dla każdej iteracji pętli.
- Po zakończeniu pętli narysuj wykres obu wielkości - błędów uczących i testowych dla kolejnych wartości N. Wskaż, jaka liczba neuronów jest optymalna dla danego zbioru danych, tj. przy jakiej liczbie neuronów błąd na części testowej jest najmniejszy.
- Naucz ostatecznie sieć neuronową już na całym zbiorze danych (a nie tylko na części uczącej), podając optymalną liczbę neuronów N.

Ćwiczenie 4.5 Oszacuj błąd popełniany przez sieć neuronową. Aby wyznaczyć w sposób dokładny błąd popełniany przez finalną nauczoną sieć względem aproksymowanej funkcji na całej dziedzinie, należałoby np. wyliczyć całkę z bezwzględnej różnicy obu funkcji (lub kwadratu różnicy) i podzielić wynik przez miarę dziedziny (tu: π^2). Nie będzie trzeba tego robić, natomiast trzeba oszacować ten błąd poprzez przybliżenie ww. całki sumą o odpowiednio dużej liczbie składników. Korzystając z programu napisanego w ćwiczeniu 4.3, należy pobrać z funkcji dodatkowy duży zbiór próbek (np. o rozmiarze rzędu 10^4) i na jego podstawie należy policzyć średni błąd bezwzględny pomiędzy oczekiwanymi wartościami y, a odpowiedziami dostarczanymi przez sieć neuronową dla testowych punktów (x_1, x_2).

5. Klasyfikacja bayesowska

Poza sieciami neuronowymi istnieje wiele innych metod pozwalających rozwiązywać problemy klasyfikacji. Patrząc z perspektywy matematycznej, metody te można w ogólności pogrupować na metody o motywacji: geometrycznej (m.in. algorytm SVM[1]) lub probabilistycznej — czyli opartej na rachunku prawdopodobieństwa (m.in. naiwny klasyfikator Bayesa), lub mieszanej (m.in. drzewa decyzyjne CART, regresja logistyczna). Niniejszy rozdział dotyczy drugiej spośród tych grup.

5.1 Elementy rachunku prawdopodobieństwa

We wszystkich podejściach probabilistycznych elementarną rolę odgrywają ***prawdopodobieństwa warunkowe***. W ramach krótkiego przypomnienia rozpoczynamy od omówienia tego pojęcia oraz kilku innych z nim powiązanych. Początkowo będziemy mówili o zdarzeniach losowych, stopniowo przechodząc do kontekstu zmiennych losowych i zbiorów danych.

[1] ang. *Support Vector Machines*

5.1.1 Prawdopodobieństwo warunkowe

Niech A i B oznaczają podzbiory pewnej przestrzeni zdarzeń Ω, tj.: $A, B \subset \Omega$. Prawdopodobieństwo wystąpienia zdarzenia A, pod warunkiem że zaszło zdarzenie B, oblicza się następujacym wzorem:

$$P(A|B) = \frac{P(A \cap B)}{P(B)}, \tag{5.1}$$

gdzie $P(B) > 0$. Innymi słowy, wśród zdarzeń elementarnych wspierających zdarzenie B patrzymy, jak często zachodzi także zdarzenie A, a zatem prawdopodobieństwo warunkowe $P(A|B)$ to iloraz miary przecięcia tych zdarzeń — $A \cap B$ (lub inaczej ich części wspólnej) w stosunku do miary zdarzenia B. W kontekście uczenia maszynowego lub eksploracji danych, możemy pytać np. o prawdopodobieństwo wystąpienia pewnej choroby pod warunkiem ustalonej płci (lub odwrotnie), prawdopodobieństwo, że grzyb jest trujący pod warunkiem cechy blaszkowatość itp. Ponadto zarówno przed jak i za kreską warunkowania możemy rozpatrywać koniunkcje pewnych zdarzeń (co będzie oznaczane symbolem $\cap$ lub krócej przecinkiem). Warto nadmienić, że wspomaniane prawdopodobieństwa są zwykle utożsamiane z odpowiednimi częstościami odczytywanymi z tabelki z danymi, które badamy.

W niektórych zadaniach lub wyprowadzeniach przydatne mogą być dodatkowo poniższe przekształcenia manipulujące wzorem na prawdopodobieństwo warunkowe:

– przenoszenie zdarzenia B za kreskę warunkowania —

$$P(A,B|C) = \frac{P(A,B,C)}{P(C)} = \frac{P(A,B,C) \cdot P(B,C)}{P(C) \cdot P(B,C)} = P(A|B,C)P(B|C). \tag{5.2}$$

– przenoszenie zdarzenia B przed kreskę warunkowania —

$$P(A|B,C) = \frac{P(A,B,C)}{P(B,C)} = \frac{P(A,B,C) \cdot P(C)}{P(B,C) \cdot P(C)} = \frac{P(A,B|C)}{P(B|C)}. \tag{5.3}$$

5.1.2 Niezależność zdarzeń

W ramach rachunku prawdopodobieństwa istnieje pojęcie ***niezależności zdarzeń*** (definicja przedstawiona poniżej). Pojęcie to niesie ważne konsekwencje dla uczenia maszynowego w ogólności, a w szczególności dla klasyfikacji bayesowskiej.

Definicja 5.1.1 — niezależność zdarzeń. Mówimy, że zdarzenia A i B są niezależne (piszemy $A \perp B$), wtedy i tylko wtedy, gdy prawdopodobieństwo ich

iloczynu (wspólnego wystąpienia) jest równe iloczynowi prawdopodobieństw:

$$P(A \cap B) = P(A) \cdot P(B). \tag{5.4}$$

Jeżeli $A \perp B$, to możemy oczekiwać, że w odpowiednio dużej populacji zdarzenie A będzie pojawiało się z taką samą częstością w całej populacji jak i warunkowo w zdarzeniu B, oraz odwrotnie — B w przybliżeniu tak samo często w całej populacji, jak i w A. Należy także zwrócić uwagę, że jezeli $A \perp B$, to:

$$P(A|B) = \frac{P(A \cap B)}{P(B)} = \frac{P(A) \cdot P(B)}{P(B)} = P(A). \tag{5.5}$$

Innymi słowy warunek mówiący o tym, że zaszło zdarzenie B nie wnosi dodatkowej informacji, która pomagałaby we wnioskowaniu o prawdopodobieństwie zajścia zdarzenia A.

Z drugiej strony, jeżeli pewne zdarzenia *nie* są niezależne, to ich występowanie razem ma inne prawdopodobieństwo (częstość) niż iloczyn prawdopodobieństw. Domniemujemy wówczas korelacji — czyli istnienia pewnej przyczyny, która te zdarzenia „wiąże” lub „odpycha”.

Przejdźmy na chwilę do ogólniejszego kontekstu zmiennych losowych (zamiast zdarzeń). Rozważmy dla przykładu zmienne losowe: *wzrost człowieka* (H) o dyskretnych wartościach $\{\mathrm{m}, \acute{\mathrm{s}}, \mathrm{d}\}$ odpowiednio o znaczeniu mały, średni, duży, oraz *kolor oczu* (C) o wartościach $\{\mathrm{z}, \mathrm{n}, \mathrm{b}, \mathrm{s}\}$ reprezentujących popularne kolory (zielony, niebieski, brązowy, szary). Aby rozpatrywane zmienne te były niezależne, wzór (5.4) musiałby zachodzić dla wszystkich możliwych podstawień par wartości do tych zmiennych, tj.:

$$\forall_{h \in \{\mathrm{m},\acute{\mathrm{s}},\mathrm{d}\}} \forall_{c \in \{\mathrm{z},\mathrm{n},\mathrm{b},\mathrm{s}\}} \quad P(H = h \cap C = c) = P(H = h) \cdot P(C = c). \tag{5.6}$$

Rozstrzygnięcie, czy dla rozpatrywanego przykładu powyższy zapis jest prawdziwy, wymagałoby dokładniejszego sprawdzenia. Niemniej warto sobie uświadomić, że można z łatwością wskazać wiele przykładów zmiennych, które *nie* są niezależne. Przykłady: płeć i wzrost człowieka (mężczyźni są statystycznie wyżsi od kobiet), wzrost i waga człowieka (ludzie wyżsi są statystycznie cięższi), cena paliwa i koszt pewnej usługi transportowej itd.

5.1.3 Prawdopodobieństwo całkowite

Ważnym pojęciem na drodze do wyprowadzenia klasyfikatora bayesowskiego jest ***prawdopodobieństwo całkowite***. Przedstawimy to pojęcie w formie poniższego twierdzenia.

Twierdzenie 5.1.1 Dla każdego rozbicia przestrzeni zdarzeń Ω na rozłączne podzbiory $B_1, B_2, \ldots, B_n$ (każdy o dodatniej mierze prawdopodobieństwa), tj:

$$\begin{aligned} \bigcup_{i=1}^{n} B_i &= \Omega, \\ \forall i \neq j \quad B_i \cap B_j &= \emptyset, \\ \forall i \quad P(B_i) &> 0, \end{aligned}$$

prawdopodobieństwo całkowite dowolnego zdarzenia A możemy obliczać wg wzoru:

$$\begin{aligned} P(A) &= P(A|B_1)P(B_1) + P(A|B_2)P(B_2) + \cdots + P(A|B_n)P(B_n) \\ &= \sum_{i=1}^{n} P(A|B_i)P(B_i). \end{aligned} \tag{5.7}$$

Dowód. Idąc od prawej strony wzory (5.7) pokażemy, że jest ona równa lewej.

$$\begin{aligned} \sum_{i=1}^{n} P(A|B_i)P(B_i) &= \sum_{i=1}^{n} \frac{P(A \cap B_i)}{P(B_i)} P(B_i) = \sum_{i=1}^{n} P(A \cap B_i) \\ &= P\left(\bigcup_{i=1}^{n} A \cap B_i\right) = P\left(A \cap \bigcup_{i=1}^{n} B_i\right) \\ &= P(A \cap \Omega) = P(A). \end{aligned}$$

Przejście z linii pierwszej do drugiej jest prawdziwe, ponieważ zbiory B_i są parami rozłączne, a więc rozłączne są również zbiory $A \cap B_i$, i w takim przypadku zachodzi własność mówiąca, że suma prawdopodobieństw jest równa prawdopodobieństwu sumy. ■

Z myślą o wzorze na prawdopodobieństwo całkowite rozważmy następujące dwa szkolne zadania. Pomogą one zrozumieć wyprowadzenie naiwnego klasyfikatora bayesowskiego, które przedstawimy w kolejnym punkcie.

1. Trzy fabryki produkują żarówki. Prawdopodobieństwo zdarzenia polegającego na tym, że wyprodukowana żarówka będzie świeciła dłużej niż 5 lat, wynoszą dla tych fabryk odpowiednio: 0.9, 0.8, 0.7. Prawdopodobieństwa napotkania na rynku żarówek z poszczególnych fabryk wynoszą odpowiednio: 0.3, 0.5, 0.2. Jakie jest prawdopodobieństwo, że losowo zakupiona żarówka będzie świeciła dłużej niż 5 lat?
2. Jeżeli wiemy, że pewna losowo zakupiona żarówka świeciła dłużej niż 5 lat, to jakie jest prawdopodobieństwo, że pochodzi ona z drugiej fabryki?

Pierwsze zadanie sprowadza się oczywiście do bezpośredniego zastosowania wzoru (5.7). Wystarczają podstawienia $P(A|B_1) = 0.9$, $P(A|B_2) = 0.8$, $P(A|B_3) = 0.7$, oraz $P(B_1) = 0.3$, $P(B_2) = 0.5$, $P(B_3) = 0.2$. Drugie zadanie to niejako zadanie odwrotne, pytające o $P(B_2|A)$. Podchodząc ogólniej, wyprowadźmy wzór na $P(B_i|A)$:

$$\begin{aligned} P(B_i|A) &= \frac{P(B_i \cap A)}{P(A)} \\ &= \frac{P(B_i \cap A)}{P(A|B_1)P(B_1) + \cdots + P(A|B_n)P(B_n)} \\ &= \frac{P(B_i \cap A)\frac{P(B_i)}{P(B_i)}}{P(A|B_1)P(B_1) + \cdots + P(A|B_n)P(B_n)} \\ &= \frac{P(A|B_i)P(B_i)}{P(A|B_1)P(B_1) + \cdots + P(A|B_n)P(B_n)}. \end{aligned} \tag{5.8}$$

Jak wskazuje ostateczny wzór, patrzymy na udział i-tego składnika w całej sumie obliczanej wg prawdopodobieństwa całkowitego.

5.2 Naiwny klasyfikator Bayesa

5.2.1 Założenie naiwne

Naiwny klasyfikator Bayesa (NBC — ang. *Naive Bayes Classifier*) to klasyfikator probabilistyczny z dołożonym tzw. ***założeniem naiwnym***, które mówi, że zmienne wejściowe są niezależne warunkowo w klasach decyzyjnych, tzn.:

$$\forall y \, \forall i \neq j \qquad X_i|Y = y \perp X_j|Y = y. \tag{5.9}$$

Oczywiście powyższe założenie rzadko kiedy jest spełnione dla rzeczywistych danych (lub wręcz prawie nigdy nie jest spełnione), co silnie akcentuje nazwa klasyfikatora. Niemniej, fakt ten nie przeszkadza w używaniu NBC w praktyce, i co więcej okazuje się, że klasyfikator ten sprawdza się bardzo dobrze dla wielu problemów.

Założenie naiwne ma ważny walor matematyczny, ponieważ pozwala na zastąpienie *prawdopodobieństwa iloczynu* pewnych zdarzeń *iloczynem prawdopodobieństw*. Niesie to ważne konsekwencje obliczeniowe, dzięki którym po pierwsze realizacja NBC jest w ogóle możliwa, a po drugie NBC radzi sobie dobrze z dużą liczbą zmiennych (cech, atrybutów) — mogą być ich setki czy nawet tysiące, i nie cierpi na tzw. *przekleństwo wymiarowości*[2]. Mówiąc dokładniej wraz ze wzro-

[2]Przekleństwo wymiarowości (ang. *curse of dimensionality*) — zjawisko występując w niektórych algorytmach uczenia maszynowego a także w metodach aproksymacji, polegające na tym, że złożoność opracowywanego modelu pewnego zjawiska (np. liczba parametrów, które należy dobrać) skaluje się wykładniczo wraz z liczbą zmiennych (cech, atrybutów) opisujących to zjawisko.

stem liczby zmiennych wejściowych złożoność (obliczeniowa i pamięciowa) NBC skaluje się liniowo, a nie wykładniczo.

5.2.2 NBC ze zmiennymi dyskretnymi

Przejdziemy teraz do wyprowadzenia najważniejszego wzoru pozwalającego obliczać odpowiedź NBC. Wzór ten ma dwa warianty — dyskretny i ciągły — zależne od tego, w jaki sposób potraktujemy zmienne wejściowe pojawiające się w danym problemie. Rozpoczniemy od bardziej intuicyjnego wariantu dyskretnego naiwnego klasyfikatora Bayesa, czyli przyjmiemy, że wszystkie zmienne są właśnie dyskretne (lub inaczej: skokowe, wyliczeniowe, kategoryczne), jak np. płeć, kolor oczu, wykształcenie, wystąpienie choroby, marka samochodu itp. Jeżeli któraś ze zmiennych nie jest dyskretna (np. wzrost, waga, prędkość, temperatura), a chcielibyśmy jej użyć, to istnieją różne techniki dokonujące dyskretyzacji takiej zmiennej, czyli zamiany jej ciągłych wartości na dyskretne (np. wzrost mały, średni, duży) z częściowo utratą informacji.

Przypuśćmy, że do dyspozycji jest pewien zbiór danych uczących z dyskretnymi zmiennymi wejściowymi X_i, $i = 1,\ldots,n$, oraz z wyróżnioną zmienną decyzyjną Y (zawierającą etykiety klas decyzyjnych). Przypuśćmy, że rozróżniamy K klas decyzyjnych, oznaczonych np. kolejnymi numerami naturalnymi $\{1,2,\ldots,K\}$. Zwyczajowo tego typu zbiór przedstawia się w formie tabelki, gdzie wierszami pisane są przykłady uczące[3] zaś kolumami zmienne (cechy, atrybuty), patrz schemat przedstawiony w tab. 5.1.

Tabela 5.1: Poglądowy schemat tabelki reprezentującej dyskretny zbiór uczący z wyróżnioną zmienną decyzyjną. Przykłady uczące pisane wierszami, zmienne kolumnami.

X_1	X_2	$\cdots$	X_n	Y
3	1	$\cdots$	2	1
2	5	$\cdots$	4	2
1	4	$\cdots$	2	2
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

Na podstawie tabelki uczącej znamy rozkłady prawdopodobieństwa (tak naprawdę rozkłady częstości) *wektorów* wejściowych w poszczególnych klasach, tj. $X = x|Y = y$. Przypuśćmy, że mamy za zadanie sklasyfikować pewien nowo przychodzący obiekt (wektor) postaci $x = (x_1, x_2, \ldots, x_n)$, gdzie x_i reprezentują

[3] inne możliwe nazwy: próbki, obserwacje, rekordy, punkty danych

konkretne wartości, np. $x = (2,3,\ldots,1)$. A zatem, chcemy wyznaczyć taką etykietę klasy (lub numer klasy) — y^*, która jest najbardziej prawdopodobna dla podanego wektora wejściowego x, co można zapisać jako:

$$y^* = \underset{y\in\{1,\ldots,K\}}{\arg\max}\, P(Y = y|X = x). \tag{5.10}$$

Zgodnie z twierdzeniem Bayesa o prawdopodobieństwie całkowitym, możemy $P(Y = y|X = x)$ rozpisać jako:

$$\begin{aligned} P(Y = y|X = x) &= \frac{P(X = x|Y = y)P(Y = y)}{P(X)} \\ &= \frac{P(X = x|Y = y)P(Y = y)}{P(X = x|Y = 1)P(Y = 1) + \cdots + P(X = x|Y = K)P(Y = K)}. \end{aligned} \tag{5.11}$$

Warto tu zwrócić uwagę, że mianownik w powyższym wzorze jest stały i niezależny od y, dla którego badamy $P(Y = y|X = x)$. A zatem możemy zignorować mianownik przy podejmowaniu decyzji o najbardziej prawdopodobnej klasie, innymi słowy zachodzi:

$$y^* = \underset{y\in\{1,\ldots,K\}}{\arg\max}\, P(Y = y|X = x) = \underset{y\in\{1,\ldots,K\}}{\arg\max}\, P(X = x|Y = y)P(Y = y). \tag{5.12}$$

Rozpiszmy pierwszy powyższy czynnik, wprowadzając założenie naiwne (przejście z linii pierwszej do drugiej):

$$\begin{aligned} P(X = x|Y = y) &= P(X_1 = x_1 \cap X_2 = x_2 \cap \cdots \cap X_n = x_n|Y = y) \\ &= P(X_1 = x_1|Y = y)P(X_2 = x_2|Y = y)\cdots P(X_n = x_n|Y = y) \\ &= \prod_{j=1}^{n} P(X_j = x_j|Y = y). \end{aligned} \tag{5.13}$$

A zatem, wychodząc od (5.12), interesujący nas końcowy **wzór dla wariantu dyskretnego**, pozwalający przyporządkować obiektowi $x = (x_1, x_2, \ldots, x_n)$ najbardziej prawdopodobną klasę, przyjmuje postać:

$$y^* = \underset{y\in\{1,\ldots,K\}}{\arg\max} \prod_{j=1}^{n} P(X_j = x_j|Y = y)P(Y = y). \tag{5.14}$$

Prosty przykład obliczeń dla NBC ze zmiennymi dyskretnymi

W ramach przećwiczenia działania NBC i wzoru (5.14) rozważmy prosty szkolny przykład. Przypuśćmy, że chcemy zastosować NBC w celu rozpoznawania (lub

przewidywania) nadciśnienia tętniczego krwi u ludzi po 40 roku życia. Rozpoznawanie chcemy oprzeć na trzech zmiennych wejściowych (cechach): płci, aktywności sportowej, paleniu. Przypuśćmy dalej, że do dyspozycji jest następująca tabelka z oznakowanymi przykładami uczącymi (uwaga: dane zostały wymyślone na potrzeby przykładu) — tab. 5.2 — czyli takimi, dla których znamy zarówno wektory cech wejściowych jak i etykietę klasy, ponieważ ta została np. określona przez lekarza. Dla uproszczenia każda zmienna X_i przyjmuje dwie możliwe wartości.

Tabela 5.2: Sztuczne dane dla problemu rozpoznawania (przewidywania) nadciśnienia tętniczego krwi u ludzi po 40 roku życia.

	X_1 — płeć	X_2 — sport	X_3 — palenie	Y — nadciśnienie
1	M	$-$	$+$	$+$
2	M	$+$	$+$	$-$
3	K	$-$	$-$	$-$
4	M	$+$	$+$	$+$
5	K	$+$	$-$	$-$
6	K	$+$	$-$	$-$
7	K	$-$	$-$	$+$
8	K	$+$	$-$	$+$
9	M	$-$	$+$	$+$
10	M	$+$	$-$	$+$
11	K	$-$	$-$	$-$
12	M	$-$	$-$	$+$
13	K	$-$	$-$	$-$
14	K	$+$	$-$	$-$
15	M	$-$	$+$	$+$
16	K	$-$	$+$	$+$

Uczenie NBC w wariancie dyskretnym polega tak naprawdę na wyznaczeniu i zapamiętaniu (w pewnej strukturze danych, np. w tablicy lub słowniku) wszystkich możliwych prawdopodobieństw, które mogą być potrzebne jako czynniki we wzorze (5.14). Utożsamiając prawdopodobieństwa z częstościami występującymi w tab. 5.2,

byłyby to następujący zestaw liczb:

$$P(Y=-)=7/16$$

$$P(X_1=M|Y=-)=1/7 \qquad P(X_1=K|Y=-)=6/7$$
$$P(X_2=-|Y=-)=3/7 \qquad P(X_2=+|Y=-)=4/7$$
$$P(X_3=-|Y=-)=6/7 \qquad P(X_3=+|Y=-)=1/7$$

$$P(Y=+)=9/16$$

$$P(X_1=M|Y=+)=6/9 \qquad P(X_1=K|Y=+)=3/9$$
$$P(X_2=-|Y=+)=6/9 \qquad P(X_2=+|Y=+)=3/9$$
$$P(X_3=-|Y=+)=4/9 \qquad P(X_3=+|Y=+)=5/9$$

Sklasyfikujemy teraz dwa przykładowe nowo przychodzące obiekty: $(M,-,+)$ oraz $(K,-,-)$. Wzór (5.14) nakazuje nam „przejść" po wszystkich klasach decyzyjnych, dla każdej z nich obliczyć odpowiedni iloczyn prawdopodobieństw, i wreszcie wybrać jako odpowiedź tę klasę, dla której iloczyn jest największy.

Ustalając na chwilę klasę $y=-$, interesujący nas iloczyn prawdopodobieństw dla obiektu $(M,-,+)$ to

$$\begin{aligned}&P(X_1=M|Y=-)\cdot P(X_2=-|Y=-)\cdot P(X_3=+|Y=-)\cdot P(Y=-)\\&\quad=\frac{1}{7}\cdot\frac{3}{7}\cdot\frac{1}{7}\cdot\frac{7}{16}=\frac{21}{5488}\approx 0.0038265,\end{aligned}$$

zaś ustalając klasę $y=+$ analogiczny iloczyn to

$$\begin{aligned}&P(X_1=M|Y=+)\cdot P(X_2=-|Y=+)\cdot P(X_3=+|Y=+)\cdot P(Y=+)\\&\quad=\frac{6}{9}\cdot\frac{6}{9}\cdot\frac{5}{9}\cdot\frac{9}{16}=\frac{1620}{11664}\approx 0.1388889.\end{aligned}$$

Jako, że druga z powyższych liczb jest większa, odpowiedzią NBC jest w tym przypadku $y^*=+$.

Warto zwrócić uwagę, że powyżej obliczone dwie wartości nie stanowią miar prawdopodobieństwa i nie sumują się do jedności. Powodem jest wspomniane wcześniej pominięcie mianownika (patrz (5.11)), który nie ma wpływu na decyzję. Jeżeli jednak z jakiegoś powodu zależy nam na wyznaczeniu liczb, które byłyby miarami prawdopodobieństwa (np. po to aby, poznać siłę wskazania na rzecz danej klasy na tle innych), to jako wspomniany mianownik należy przyjąć sumę obliczonych iloczynów (w zgodzie z założeniem naiwnym). W rozważanym przykładzie

można by wówczas napisać:

$$P\big(Y=-|X=(M,-,+)\big)=\frac{\frac{21}{5\,488}}{\frac{21}{5\,488}+\frac{1\,620}{11\,664}}\approx 0.0268123,$$
$$P\big(Y=+|X=(M,-,+)\big)=\frac{\frac{1\,620}{11\,664}}{\frac{21}{5\,488}+\frac{1\,620}{11\,664}}\approx 0.9731877.$$

Postępując analogicznie dla drugiego obiektu $(K,-,-)$, można przekonać się, że interesujące nas iloczyny wynoszą odpowiednio $756/5\,488$ i $648/11\,664$, które po znormalizowaniu do prawdopodobieństw przełożyłyby sie na:

$$P\big(Y=-|X=(K,-,-)\big)=\frac{\frac{756}{5\,488}}{\frac{756}{5\,488}+\frac{648}{11\,664}}\approx 0.7094635,$$
$$P\big(Y=+|X=(K,-,-)\big)=\frac{\frac{648}{11\,664}}{\frac{756}{5\,488}+\frac{648}{11\,664}}\approx 0.2905365.$$

Patrząc ponownie na główny wzór (5.14), warto zwrócić uwagę na rolę, jaką pełni w nim czynnik $P(Y=y)$ nazywamy *prawdopodobieństwem a priori* klasy. W rozważanym powyżej przykładzie rozkład prawdopodobieństw a priori dla klas wynosił: $P(Y=-)=7/16$, $P(Y=+)=9/16$. Daje to pewną przewagę klasie $Y=+$ przy obliczaniu odpowiedzi klasyfikatora, ale jest to przewaga drobna — rozkład klas jest bliski równomiernemu. Jeżeli natomiast rozważylibyśmy problem wykrywania pewnego bardzo rzadkiego zjawiska (np. pożaru w monitorowanym obiekcie, obecności rzadkiego wirusa w całej populacji itp.), to rozkład a priori byłby daleki od równomiernego[4] i rozpoznanie klasy o małym $P(Y=y)$ musiałoby się wiązać z wysokimi wartościami wielu innych czynników występujących we wzorze (5.14). Omawiany tu aspekt warto też odnieść do tzw. *klasyfikatora bezregułowego* (ang. *zero-rule classifier*). Nakazuje on odpowiadać zawsze klasą najczęstszą w rozkładzie a priori, nie zwracając uwagi na cechy badanego obiektu. Klasyfikator ten należy traktować jako punkt odniesienia, gdy zastanawiamy się, na ile dobry klasyfikator uzyskaliśmy dla naszego problemu. Dla przykładu powiedzmy, że zajmujemy się problemem wykrywania wiadomości e-mail będących spamem i nasz zbiór uczący, zebrany np. wśród pracowników uczelni, wskazuje na rozkład a priori: $P(Y=\text{nie-spam})=0.2$, $P(Y=\text{spam})=0.8$. Wówczas klasyfikator bezregułowy klasyfikowałby „na ślepo” wszystkie przychodzące wiadomości jako spam. W takiej sytuacji od dowolnego opracowanego klasyfikatora — bayesowskiego, sieci neuronowej, drzewa CART, maszyny SVM itd. — wymagamy, aby miał on dokładność rozpoznawania powyżej 0.8 (mowa tu o dokładności zmierzonej na

[4] Mówimy wówczas o tzw. danych niezrównoważonych (ang. *imbalanced data*).

zbiorze testowym nie widzianym podczas uczenia). W przeciwnym razie nie byłoby żadnego zysku z uczenia maszynowego i stosowania klasyfikatora.

Dowolny opracowany klasyfikator powinien pod względem dokładności przewyższać klasyfikator bezregułowy (ang. *zero-rule classifier*).

Złożoność NBC ze zmiennymi dyskretnymi

Zastanówmy się teraz nad złożonością naiwnego klasyfikatora bayesowskiego. Jeżeli chodzi o złożoność obliczeniową związaną z samym wyznaczaniem odpowiedzi wg wzoru (5.14), to łatwo stwierdzić, że jest ona klasy $O(Kn)$ — czyli liniowa ze względu liczbę klas i (co ważniejsze) liniowa na liczbę atrybutów, zakładając, że każdy potrzebny nam czynnik możemy odczytać w czasie stałym $O(1)$ z tablicy (lub słownika), gdzie są one przechowywane. Jeżeli chodzi o złożoność pamięciową związaną z przechowaniem tejże struktury danych, to można ją przedstawić jako $O(K+Kn\bar{q}) \sim O(Kn\bar{q})$), gdzie $\bar{q}$ oznacza średnią liczbę wartości, które osiągają przyjęte zmienne[5]. Rozważmy teraz złożoność obliczeniową uczenia NBC — czyli, ile czasu wymaga przygotowanie wyżej wspomnianej struktury danych z prawdopodobieństwami. Potrzebujemy wyznaczyć K prawdopodobieństw postaci $P(Y=y)$ oraz $n\bar{q}K$, prawdopodobieństw postaci $P(X_i=v|Y=y)$, gdzie v to pewna wartość. A zatem pozornie (i przy niestarannym podejściu do implementacji) mogłoby wydawać się, że złożoność obliczeniowa jest rzędu $O(mn\bar{q}K)$, gdzie pierwszy czynnik m byłby związany z przebiegiem po przykładach uczących. Jest to jednak błędne wrażenie, ponieważ nie potrzebujemy skanować wielokrotnie tabelki z danymi dla każdej możliwej wartości v i dla każdej klasy y występującej w $P(X_j=v|Y=y)$, zaś wystarcza dokładnie jeden przebieg po danych. Dla przykładu, biegnąc po pierwszym wierszu tab. 5.2, napotykamy kolejno wartości (symbole): M, $-$, $+$. Wiedząc, że wiersz ten jest przypisany do klasy $y=+$, wystarcza, abyśmy podnieśli o jeden odpowiednie liczniki związane ze zdarzeniami: $X_1=M|Y=+$, $X_2=-|Y=+$, $X_3=+|Y=+$. Po zakończeniu przebiegu po wszystkich wierszach liczniki należy zamienić na prawdopodobieństwa, wykonując dzielenia dla poszczególnych klas zgodnie ze wzorem na prawdopodobieństwo warunkowe (5.1). A zatem złożoność obliczeniowa uczenia NBC jest tylko rzędu $O(mn)$.

Złożoność pamięciowa dyskretnego NBC: $O(Kn\bar{q})$.
Złożoność obliczeniowa uczenia dyskretnego NBC: $O(mn)$.
Złożoność obliczeniowa wyznaczenia odpowiedzi dyskretnego NBC: $O(Kn)$.

[5]Np. dla zmiennych *płeć* o wartościach $\{M,K\}$ oraz *kolor oczu* np. o wartościach $\{zielone, niebieskie, brzowe, piwne, szare\}$ mamy średnio 3.5 wartości na zmienną.

Poprawka LaPlace'a

Oko czujnego matematyka lub programisty może zauważyć pewne niebezpieczeństwo obliczeniowe tkwiące we wzorze (5.14). Co, jeśli którykolwiek z czynników w tym wzorze byłby równy 0? Oczywiście, spowodowałoby to wyzerowanie całego wyniku niezależnie od wartości pozostałych czynników (nawet jeśli te byłyby bardzo wysokie). Byłaby to sytuacja niepożądana. Kiedy mogłoby dojść do niej? Zgodnie z tym, co powiedziano wcześniej, zwyczajowo utożsamia się prawdopodobieństwa z częstościami w zbiorze uczącym. I takie podejście nie jest błędne, jeżeli zbiór danych jest odpowiednio duży. Do sytuacji, o której mowa, mogłoby dojść wtedy, gdyby w zbiorze uczącym nie zaistniałaby realizacja pewnego zdarzenia, np. nigdy nie zaobserwowano $X_3 = 5|Y = 2$, a podczas testowania klasyfikatora pojawiłby się obiekt, dla którego trzecia cecha ma właśnie wartość 5.

Istnieją różne techniki radzenia sobie z tym niebezpieczeństwem, polegających w ogólności na *wygładzaniu* rozkładów prawdopodobieństwa (ang. *distribution smoothing*) i tym samym unikaniu skrajnych prawdopodobieństw (zarówno zer jak i jedynek). Jedną z najbardziej popularnych jest tzw. ***poprawka LaPlace'a***. Przypuśćmy, że w m próbach zaobserowaliśmy k wystąpień pewnego zdarzenia A dotyczącego zmiennej o q unikalnych wartościach. Szacując prawdopodobieństwo na podstawie częstości, powinniśmy napisać $P(A) \approx k/m$. Stosując poprawkę LaPlace'a, oszacowanie przybiera postać

$$P(A) \approx \frac{k+1}{m+q}. \tag{5.15}$$

W szczególności dla zdarzeń binarnych powyższy wzór wynosi $(k+1)/(m+2)$.

Należy mieć świadomość, że dla małych zbiorów danych poprawka LaPlace'a zwykle psuje nieznacznie dokładność uczącą klasyfikatora, czyli jego zdolność do bezbłędnego odtworzenia etykiet danych uczących. Niemniej, jednocześnie (w takich sytuacjach) poprawka ta poprawia dokładność testową, czyli zdolność do uogólniania (generalizacji) dla niewidzianych obserwacji, a na tym właśnie elemencie zależy nam w uczeniu maszynowym.

Konsekwencje założenia naiwnego

Kończąc omawianie podstawowego dyskretnego wariantu NBC, warto zwrócić jeszcze uwagę na zysk płynący z przyjętego założenia naiwnego, oraz na pewną trudność, która miałaby miejsce bez tego założenia. Zysk tyczy wspomnianej złożoności pamięciowej $O(K\,n\,\bar{q})$. Należy zauważyć, że bez założenia naiwnego musielibyśmy przechowywać w pamięci prawdopodobieństwa wystąpień wszystkich unikalnych *wektorów* cech (zamiast wartości pojedynczych cech) pod warunkiem poszczególnych klas. Oznaczałoby to złożoność $O(K\,\bar{q}^n)$, która w wielu przypadkach byłaby niemożliwa do osiągnięcia ze względu na wykładniczą zależność względem n

(przekleństwo wymiarowości). Dodatkowa trudność polega na tym, że nawet jeżeli dla odpowiednio małego problemu istniałaby możliwość zapamiętania $O(K\bar{q}^n)$ prawdopodobieństw dla wektorów, to wiele z nich byłyby błędnie równe zero z uwagi na braki realizacji wszystkich możliwych wartości wektorowych.

5.2.3 NBC ze zmiennymi ciągłymi

Jeżeli chcielibyśmy używać naiwnego klasyfikatora Bayesa, pracując na zmiennych ciągłych (wzrost, temperatura itp.) w sposób bezpośredni, tzn. nie dyskretyzując ich, to wzór (5.14) nie pozwala nam na to. Operuje on bowiem na prawdopodobieństwach pewnych zdarzeń rozumianych (mówiąc nieformalnie) w sposób gruboziarnisty, np. płeć = kobieta, kolor oczu = szary. Co, jeśli klasyfikacji ma podeglać człowiek np. o wzroście 188.7 cm? Zwróćmy również uwagę, że poza powyższymi oczywistymi przykładami w wielu problemach istnieją zmienne o charakterze dyskretnym z natury rzeczy, a mimo to wolelibyśmy je traktować w sposób ciągły, np. intensywność piskela o zbiorze wartości $\{0,1\ldots,255\}$.

Warto przypomnieć, że w rachunku prawdopodobieństwa dla zmiennych ciągłych rozróżniamy zwyczajowo dwie funkcje związane z rozkładem: funkcję *gęstości* rozkładu prawdopodobieństwa (PDF — ang. *probability density function*) oraz *dystrybuantę* zwaną także *kumulantą* (CDF — ang. *cumulative distribution function*). Wartości funkcji gęstości w punkcie nie mają jako takiego sensu probabilistycznego, a dopiero całki funkcji gęstości (obliczone nad przedziałami lub innymi zbiorami) stanowią miary prawdopodobieństwa pewnych zdarzeń. Np. jeżeli p oznacza funkcję gęstości pewnej skalarnej zmiennej X, to prawdopodobieństwa zdarzenia, że wartość (realizacja) tej zmiennej należy do przedziału $[a,b]$, możemy obliczyć następująco:

$$P(a \leqslant X \leqslant b) = \int_a^b p(x)\,dx. \tag{5.16}$$

Oczywiście, prawidłowe funkcje gęstości całkują się nad całą dziedziną do jedynki, czyli np. dla gęstości jednowymiarowych mamy $\int_{-\infty}^{\infty} p(x)\,dx = 1$. Z kolei wartości funkcji dystrybuanty w punkcie mają sens probabilistyczny. Jeżeli oznaczyć dystrybuantę przez F, to:

$$F(a) = P(X \leqslant a) = \int_{-\infty}^{a} p(x)\,dx. \tag{5.17}$$

Można zapisać następujące związki pomiędzy gęstością a dystrybutantą:

– różniczka dystrybuanty = gęstość · przyrost:

$$dF(x) = p(x)\,dx, \tag{5.18}$$

– całka nieoznaczona z funkcji gęstości = dystrybuanta + stała:

$$\int p(x)\,dx = F(x) + C \tag{5.19}$$

– prawdopodobieństwo jako przyrost dystrybuanty:

$$P(a \leqslant X \leqslant b) = \int_a^b p(x)\,dx = F(b) - F(a). \tag{5.20}$$

Interesujący nas **wzór dla wariantu ciągłego** naiwnego klasyfikatora Bayesa to „kuzyn" wzoru (5.14), w którym w miejsce prawdopodobieństw wpisujemy wartości warunkowych funkcji *gęstości* w punkcie (z wyjątkiem prawdopodobieństw a priori klas — te pozostają bez zmian):

$$y^* = \underset{y\in\{1,\ldots,K\}}{\arg\max} \prod_{j=1}^{n} p_j(x_j|Y=y)P(Y=y). \tag{5.21}$$

Należy być świadomym następujących ograniczeń związanych ze wzorem (5.21):

1. zwracana przezeń wartość nie powinna być interpretowana jako prawdopodobieństwo, nawet po normalizacji, ze względu na mieszane czynniki p oraz P o różnym sensie probabilistycznym (wzór jedynie „przypomina" iloczyn prawdopodobieństw),
2. nadal w mocy jest założenie naiwne — przy wyprowadzaniu wzoru (5.21) (które pominęliśmy) gęstości *łącznych* warunkowych rozkładów prawdopodobieństw $p(\mathbf{x}|Y=y)$, gdzie $\mathbf{x} = (x_1,\ldots,x_n))$ jest wektorem w $\mathbb{R}^n$, należy zamienić na iloczyn gęstości dla pojedynczych zmiennych $p_j(x_j|Y=y)$,
3. aby móc używać wzoru (5.21) należy wyznaczyć za pomocą wybranego podejścia *estymaty* funkcji gęstości $p_j(x_j|Y=y)$ na podstawie danych uczących (np. przyjmując, że są one zgodne z rozkładami normalnymi).

Estymaty za pomocą rozkładów normalnych

Rozkłady normalne (zwane także gaussowskimi) obserwujemy bardzo często w przyrodzie. Fakt ten można w dużej mierze można wytłumaczyć poprzez Centralne Twierdzenie Graniczne mówiące, że rozkład zmiennej losowej, która jest sumą innych niezależnych zmiennych losowych, zbiega szybko do rozkładu normalnego wraz z liczbą składników. Wiele rzeczywistych wielkości, które obserwujemy lub mierzymy, można często rozumieć właśnie jako wypadkową (lub sumę) pewnych drobnych, niskopoziomowych elementów lub przyczyn.

W związku z powyższym argumentem popularnym podejściem do realizacji ciągłego NBC jest estymowanie rozkładów zmiennych ciągłych za pomocą rozkładów *normalnych*. Oczywiście należy być świadomym, że jest to uproszczenie,

które może przekłamywać wpływ niektórych szczególnych zmiennych w obliczanym iloczynie, tzn. tych zmiennych, których rozkłady są dalekie od normalnych (chociażby rozkłady wielomodalne).

Wzór funkcji gęstości dla rozkładu normalnego jednej zmiennej ma postać

$$p(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}, \tag{5.22}$$

gdzie parametry μ i σ oznaczają odpowiednio wartość *średnią* (lub oczekiwaną) oraz *odchylenie standardowe* rozkładu. Parametry te można oszacować na podstawie skończonej próby, czyli na podstawie zbioru danych myśląc o kontekście uczenia maszynowego. Dla uproszczenia, przypuśćmy że wszystkie rozpatrywane zmienne wejściowe są ciągłe, i przypomnijmy przyjętą we wcześniejszym rozdziale notację dla zbioru danych postaci: $D = \{(\mathbf{x}_i, y_i)\}_{i=1,\ldots,m}$, gdzie $\mathbf{x}_i = (x_{i1}, x_{i2}, \ldots, x_{in}) \in \mathbb{R}^n$ są wektorami cech rzeczywistoliczbowych, zaś y_i etykietami klas. A zatem chcąc przygotować NBC w wariancie ciągłym gaussowskim, musimy wyznaczyć $2 \cdot n \cdot K$ parametrów — oznaczmy je jako μ_{jy} i σ_{jy} (z użyciem pary indeksów) — będących średnimi i odchyleniami standardowymi, dla wszystkich warunkowych rozkładów zmiennych $X_j|Y = y$, gdzie $j = 1, \ldots, n$, $y = 1, \ldots, K$. Oznaczając gęstość takiego wybranego rozkładu jako

$$p_j(x|Y = y) = \frac{1}{\sigma_{jy}\sqrt{2\pi}} e^{-\frac{(x-\mu_{jy})^2}{2\sigma_{jy}^2}}, \tag{5.23}$$

stosuje się poniższe wzory do wyznaczenia estymat odpowiednio średniej i odchylenia standardowego:

$$\mu_{jy} = \frac{1}{m} \sum_{\substack{i=1 \\ y_i=y}}^{m} x_{ij}, \tag{5.24}$$

$$\sigma_{jy} = \sqrt{\frac{1}{m-1} \sum_{\substack{i=1 \\ y_i=y}}^{m} (x_{ij} - \mu_{jy})^2}. \tag{5.25}$$

Uwaga — czynnik normalizujący $\frac{1}{m-1}$ widoczny w drugim wzorze nie jest pomyłką, a wynika z posłużenia się tzw. *estymatorem nieobciążonym*[6].

[6]Można udowodnić, że wartość oczekiwana wzoru (5.24) wzięta po wszystkich możliwych realizacjach próby m-elementowej (pochodzącej z ustalonego rozkładu) oddaje dokładne odchylenie standardowe interesującej nas zmiennej. Jednocześnie nie ma to miejsca, jeżeliby stosować naturalnie wyglądający czynnik $\frac{1}{m}$.

5.2.4 Przykłady działania NBC

Tab. 5.3 prezentuje dokładności naiwnego klasyfikatora bayesowskiego (w różnych wariantach) uzyskane dla kilku znanych benchmarkowych zbiorów danych pobranych z repozytorium UCI[7] oraz dla sztucznego zbioru „moons” (punkty danych rozłożone na płaszczyźnie w kształcie dwóch zazębiających się księżyców) wygenerowanego z użyciem pakietu `scikit-learn` języka Python.[8]

Tabela 5.3: Dokładność klasyfikatorów bayesowskich dla przykładowych zbiorów danych z repozytorium UCI.

		dyskretny NB $q=3$		dyskretny NB $q=5$		dyskretny NB $q=7$		gaussowski NB	
		dokładność		dokładność		dokładność		dokładność	
dane	pełny rozmiar, liczba klas	ucząca	testowa	ucząca	testowa	ucząca	testowa	ucząca	testowa
„moons”	100×2, 2	86.67%	84.00%	85.33%	92.00%	93.33%	100.00%	84.00%	92.00%
„wine”	178×13, 3	92.48%	100.00%	97.74%	95.56%	96.99%	100.00%	97.74%	95.56%
„spambase”	4601×57, 2	68.20%	67.51%	78.26%	77.76%	82.23%	82.19%	82.06%	83.93%
„iris”	150×4, 3	94.64%	97.37%	93.75%	92.11%	90.18%	86.84%	94.64%	97.37%
„sonar”	208×60, 2	80.13%	76.92%	88.46%	76.92%	86.54%	84.62%	73.08%	71.15%

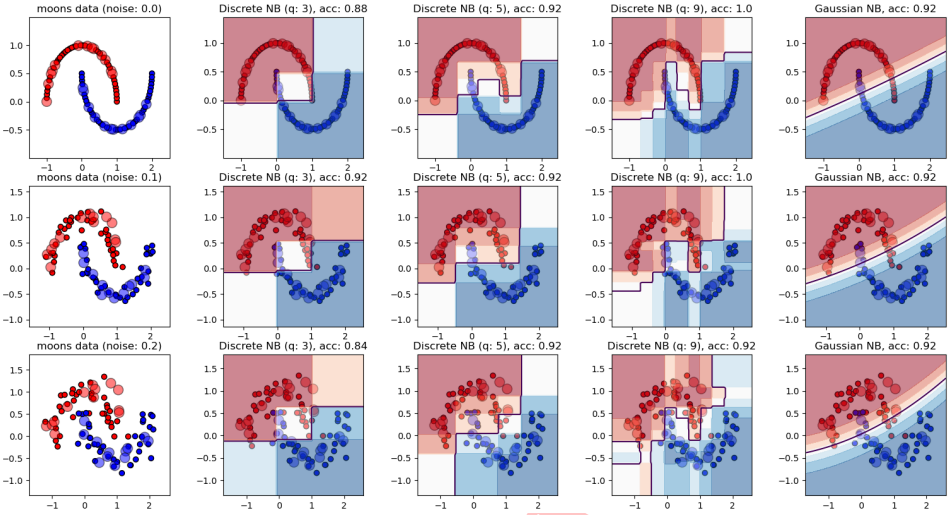
W drugiej kolumnie tabeli podano pełne rozmiary zbiorów — liczba przykładów $\times$ liczba zmiennych (cech) — oraz po przecinku liczbę klas decyzyjnych. Każdy zbiór danych został wstępnie podzielony na część uczącą i testową w proporcji 75% : 25%. Do przygotowania klasyfikatorów (czyli wyznaczenia i zapamiętania potrzebnych rozkładów prawdopodobieństwa) użyto tylko części uczącej. Tabela raportuje otrzymane procentowe dokładności dla obu części — uczącej i testowej — przy czym oczywiście tylko dokładność testowa świadczy o jakości klasyfikatora (jego zdolności do generalizacji). Klasyfikatory w wariancie dyskretnym są opatrzone parametrem q oznaczającym przyjętą na potrzeby dyskretyzacji liczbę przedziałów, na którą dzielone były zmienne ciągłe (przedziały równoszerokie).

Na rys. 5.1 pokazano przykładowe wizualizacje granic decyzyjnych wyznaczanych przez naiwne klasyfikatory bayesowskie dla zbioru „moons” (z różnymi nastawami zaszumienia). Nad wizualizacjami podano dokładności testowe (testowe punkty danych zaznaczono większymi bladymi kołami).

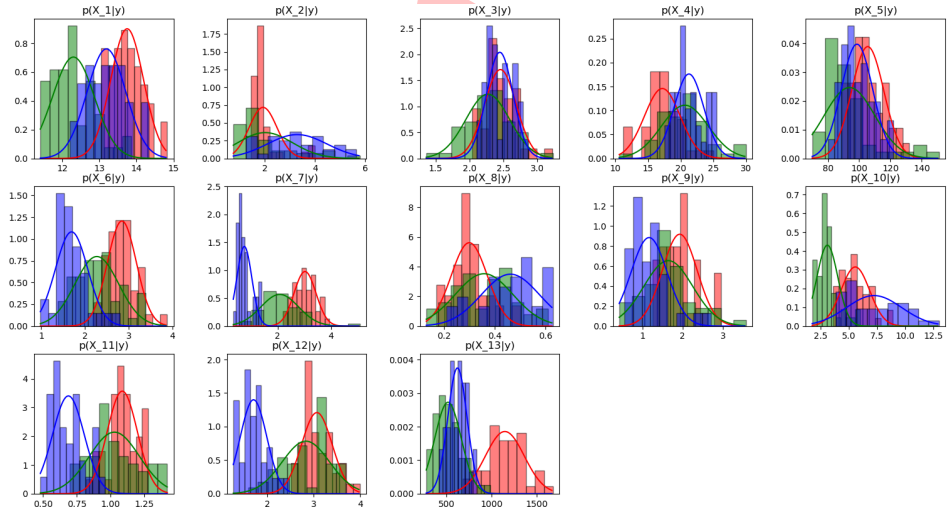
Dla lepszego zrozumienia różnic w podejściu do estymowania rozkładów pomiędzy dyskretnymi a gaussowskimi NBC przygotowano rysunki 5.2 i 5.3. Dotyczą

[7] Publiczne akademickie repozytorium zbiorów danych Uniwersytetu Kalifornijskiego w Irvine — UCI Machine Learning Repository: `https://archive.ics.uci.edu/ml/index.php`.

[8] Wyniki przedstawione w tabeli dla zbioru „moons” dotyczą wariantu tego zbioru wygenerowanego z parametrami `noise=0.1` oraz `random_state=0`. Wywołanie: `sklearn.datasets.make_moons(noise=0.1, random_state=0)`.

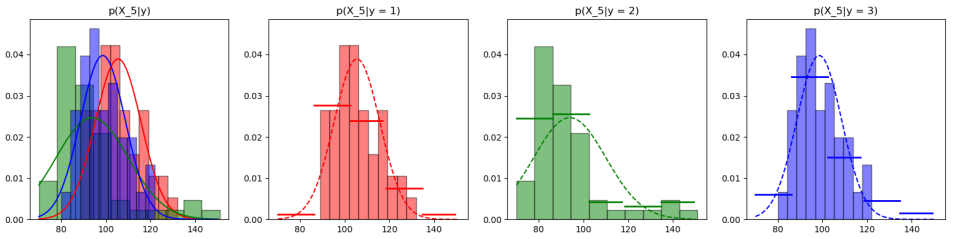


Rys. 5.1: Naiwne dyskretne klasyfikatory Bayesa dla danych „moons" generowanych z różnym zaszumieniem. Czarna granica decyzyjna odpowiada prawdopodobieństwu 1/2. Raportowane nad wykresami dokładności (acc) dotyczą testowych punktów danych zaznaczonych większymi bladymi kołami. (źródło: *opracowanie własne*)



Rys. 5.2: Histogramy i przybliżenia normalne dla warunkowych rozkładów prawdopodobieństwa zmiennych w danych „wine". (źródło: *opracowanie własne*)

one zbioru „wine" (rozpoznawanie gatunków wina) i przedstawiają: histogramy wszystkich rozkładów warunkowych, ich przybliżenia normalne, a także przybliże-



Rys. 5.3: Rozkład zmiennej nr 5 w danych „wine" — porównanie: przybliżenia normalne vs. przybliżenia kawałkami stałe (przy dyskretyzacji na 5 równoszerokich przedziałów). (źródło: *opracowanie własne*)

nia kawałkami stałe dla wybranej zmiennej będące równoważne dyskretyzacji.

5.2.5 Bezpieczeństwo numeryczne obliczeń NBC

Wzory wyznaczające odpowiedź NBC — (5.14) i (5.21) — to długie iloczyny prawdopodobieństw i / lub gęstości, czyli zazwyczaj[9] iloczyny liczb z przedziału $[0, 1]$. Mając świadomość ograniczeń *zmiennoprzecinkowych* typów liczbowych (`float` lub `double`)[10], w których powszechnie realizujemy obliczenia na komputerze, należy zauważyć, że przy odpowiednio dużej liczbie czynników iloczyn ten może wyzerować się numerycznie, pomimo że matematycznie powinien być dodatni — sytuacja niedomiaru obliczeń (ang. *underflow*). Dla przykładu, pracując na liczbach zmiennoprzecinkowych podwójnej precyzji (`double`), można łatwo przekonać się, że iloczyn zaledwie 324 czynników równych 0.1 stanie się równy *zeru* w tym typie liczbowym (niezależnie od języka programowania). A zatem implementacje NBC używające kilkuset zmiennych wejściowych mogą być narażone na obliczanie nieprawidłowych odpowiedzi dla niektórych danych wejściowych. Sytuacja ta może także mieć miejsce dla dużo mniejszej liczby zmiennych, a przy obecności istotnie małych wartości w rozkładach prawdopodobieństw.

Użyteczną „sztuczką" obliczeniową pozwalającą radzić sobie w praktyce z tym problemem jest *logarytmowanie*. Po pierwsze, zgodnie z tożsamością matematyczną, logarytm iloczynu jest równy sumie logarytmów. Po drugie, logarytm jest funkcją monotoniczną. A zatem każdy ze wzorów (5.14), (5.21) można zlogarytmować, zamieniając tym samym iloczyn na sumę, co nie wpłynie na podejmowaną decyzję ze względu na monotoniczność. Logarytmy prawdopodobieństw są liczbami ujemnymi[11], a więc czytelnik może zastanawiać się, kiedy suma odpowiednio wielu ujemnych składników również doprowadzi do niedomiaru i osiągnie war-

[9] Wartości funkcji gęstości w punkcie *nie* są oczywiście ograniczone do przedziału $[0, 1]$.
[10] Typy określone przez standard IEEE 754.
[11] niekoniecznie logarytmy funkcji gęstości

tość `-inf`. Okazuje się, że sumowanie dużo wolniej wyczerpuje dostępną precyzję mantysy (51 bitów w podwójnej precyzji), aniżeli mnożenie ułamków wyczerpuje precyzję wykładnika (11 bitów w podwójnej precyzji). Tym samym, zabieg logarytmowania istotnie podnosi bezpieczeństwo numeryczne obliczeń w naiwnym klasyfikatorze bayesowskim.

Odpowiednik wzoru (5.14) dla NBC ze zmiennymi dyskretnymi przyjmuje następującą postać po zlogarytmowaniu:

$$y^* = \underset{y\in\{1,\ldots,K\}}{\arg\max} \sum_{j=1}^{n} \log P(X_j = x_j | Y = y) + \log P(Y = y). \tag{5.26}$$

Z kolei odpowiednik wzoru (5.21) dla NBC ze zmiennymi ciągłymi modelowanymi poprzez rozkłady normalne o funkcji gęstości (5.23) przyjmuje następującą postać po zlogarytmowaniu:

$$y^* = \underset{y\in\{1,\ldots,K\}}{\arg\max} \sum_{j=1}^{n} \left(\log 1 - \log \sigma_{jy} - \log \sqrt{2\pi} + \log e^{-\frac{(x-\mu_{jy})^2}{2\sigma_{jy}^2}} \right) + \log P(Y = y) \tag{5.27}$$

$$= \underset{y\in\{1,\ldots,K\}}{\arg\max} \sum_{j=1}^{n} \left(-\log \sigma_{jy} - \frac{(x-\mu_{jy})^2}{2\sigma_{jy}^2} \right) + \log P(Y = y). \tag{5.28}$$

W ostatnim przejściu równościowym wykorzystano fakt, że $\log\sqrt{2\pi}$ jest stałą niezależną od badanej klasy y, a zatem nie wpływa na decyzję (wybrane maksimum pozostaje w tym samym miejscu).

Zamiana długich iloczynów na sumę logarytmów podnosi bezpieczeństwo numeryczne obliczeń w naiwnych klasyfikatorach bayesowskich.

5.3 Ćwiczenia laboratoryjne (Python)

E **Ćwiczenie 5.1 Napisz program realizujący NBC w wersji dyskretnej dla zbioru „wine" z repozytorium UCI** Z repozytorium UCI[12] pobierz zbiór danych o nazwie „wine" dotyczący klasyfikacji wina na podstawie składu chemicznego i zapoznaj się z nim. Zwróć uwagę, która ze zmiennych jest zmienną decyzyjną. Wczytaj dane z pobranego pliku tekstowego `wine.data` do macierzy `numpy` (wykorzystaj funkcję `numpy.genfromtxt`) i rozdziel tę macierz na dwie macierze `X` (o wymiarze 178×13) i `y` (178×1 — etykiety klas). Dyskretyzację danych „wine" można wykonać wykorzystując gotowy obiekt `KBinsDiscretizer` (z pakietu `sklearn.preprocessing`) lub samodzielnie na poziomie opracowywanej klasy NBC (liczba przedziałów, na którą dyskretyzujemy cechy oryginalnie ciągłe, powinna być parametrem nastawialnym przez użytkownika). Podziel dane na część uczącą i testową (wykorzystaj funkcję `train_test_split` z pakietu `sklearn.model_selection`). Napisz klasę reprezentującą naiwny klasyfikator Bayesa w wariancie ze zmiennymi dyskretnymi. Klasę przygotuj zgodnie z ideą biblioteki `scikit-learn` — m.in.: wykonaj dziedziczenie po klasach `BaseEstimator` i `ClassifierMixin`, przygotuj metody `fit` (uczenie) i `predict` (klasyfikowanie) oraz pomocniczo `predict_proba`. Zastanów się i zaplanuj wg własnego uznania wygodne struktury danych do przechowywania:

- rozkładu a priori klas $P(Y = y)$,
- rozkładów warunkowych $P(X_j = v|Y = y)$.

Mogą to być tablice, słowniki, listy lub odpowiednie połączenia / zagnieżdżenia tych struktur. Do tego celu potrzebne będzie także ustalenie dyskretnych dziedzin zmiennych, tj. wykrycie, jakie unikalne wartości poszczególne zmienne mogą przyjmować, np. z wykorzystaniem funkcji `numpy.unique`. Przemyśl, czy informacje o dziedzinach należy zdobywać na poziomie funkcji `fit` na podstawie danych uczących, czy też lepiej przekazać je klasyfikatorowi już podczas konstrukcji. Uwaga: w ramach tego ćwiczenia obliczanie odpowiedzi klasyfikatora (w metodach `predict_proba`, `predict`) może być realizowane zgodnie ze wzorem (5.14) tj. jako iloczyn prawdopodobieństw (bez zabiegu logarytmowania). Wyznacz dokładność otrzymanego klasyfikatora na zbiorach uczącym i testowym. Obliczenia powtórz uwzględniając poprawkę LaPlace'a (możesz do tego celu wprowadzić przełącznik w konstruktorze Twojej klasy). Zwróć uwagę, czy poprawka LaPlace'a podnosi dokładność testową dla tego zbioru danych.

E **Ćwiczenie 5.2 Napisz program realizujący NBC w wersji ciągłej dla zbioru „wine" z repozytorium UCI** Jako rozszerzenie ćwiczenia 5.1 opracuj nowy klasyfikator bayesowski (nowa klasa) realizujący klasyfikację danych z winem w wariancie ciągłym, czyli bez wykonywania dyskretyzacji danych. Zastosuj estymaty funkcji gęstości oparte na rozkładach normalnych. W szczególności zaplanuj odpowiednie struktury danych do przechowywania średnich i odchyleń standardowych dla poszczególnych gęstości warunkowych. Porównaj dokładność otrzymanego klasyfikatora z jego dyskretnym odpowiednikiem. Porównaj także zgodność działania otrzymanego klasyfikatora (Twojej implementacji) z gotową implementacją `GaussianNB` dostępną w pakiecie `sklearn.naive_bayes`.

[12] `https://archive.ics.uci.edu/ml/index.php`

E **Ćwiczenie 5.3 Opracuj NBC dla nowego zbioru danych (innego niż „wine”)** Znajdź nowy większy zbiór danych (może być z repozytorium UCI) stosowny dla zadania klasyfikacji. Zbiór powinien zawierać przynajmniej 1 000 przykładów opisanych przynajmniej 20 cechami (zmiennymi). Zgodnie z naturą tego zbioru (dane dyskretne / ciągłe) opracuj odpowiedni dla niego naiwny klasyfikator bayesowski. Przeprowadź eksperymenty, raportując otrzymaną dokładność testową. W przypadku dyskretnym sprawdź, jaki wpływ na dokładność mają poprawka LaPlace'a oraz wybór liczby przedziałów podczas dyskretyzacji zmiennych ciągłych.

E **Ćwiczenie 5.4 Zmodyfikuj opracowane implementacje NBC zapewniając bezpieczeństwo numeryczne obliczeń** Wykorzystując zabieg logarytmowania, zmodyfikuj implementacje opracowane na rzecz ćwiczeń 5.1 i 5.2, tak aby obliczanie odpowiedzi klasyfikatora było zgodne odpowiednio z wzorami (5.26) i (5.28). Wskazówka dla wariantu dyskretnego: w wybranych przez Ciebie strukturach danych możesz od razu przechowywać logarytmy prawdopodobieństw zamiast prawdopodobieństw (zmiana na poziomie funkcji `fit`); tym samym później, w trakcie obliczania odpowiedzi klasyfikatora (funkcje `predict_proba` i / lub `predict`) wystarczy samo sumowanie przechowanych wartości (logarytmowanie nie będzie potrzebne). Spróbuj zaaranżować sytuację niebezpieczną numerycznie, np. rozmnażając sztucznie liczbę cech (kolumn) w zbiorze danych, i porównaj działanie poprzednich implementacji niebezpiecznych numerycznie z nowymi (bezpiecznymi).

6. Podstawy Statystycznej Teorii Uczenia

Ostatnie trzy dekady to bardzo istotny rozwój algorytmów ***uczenia maszynowego*** czyli inaczej algorytmów uczących się z danych. Stopniowo stają się one coraz powszechniejsze w różnych dziedzinach działalności człowieka, m.in. w: medycynie, biotechnologii, widzeniu komputerowym, motoryzacji, ekonomii, technologiach produkcyjnych itp., gdzie pojawiło się dużo praktycznych aplikacji opartych na gromadzonych zbiorach danych. Istnieją pewne algorytmy, które sprawdzają się szczególnie dobrze w praktyce i są obecnie uznawane za tzw. *state-of-art* m.in.: maszyny SVM, perceptrony, klasyfikatory bayesowskie, drzewa decyzyjne, lasy losowe, boosting, głębokie sieci neuronowe. Jednakże warto jednocześnie zaznaczyć, że zgodnie z twierdzeniem Wolperta „nie ma darmowego lunchu" [Wol96] (ang. *no free lunch theorem*) żaden z algorytmów uczenia maszynowego tak naprawdę nie może zostać wyróżniony a priori, tj. przed obejrzeniem danych. Innymi słowy na pewnym konkretnym zbiorze danych może najlepiej zadziałać np. algorytm SVM, na innym sieć neuronowa, jeszcze na innym AdaBoost itd. Oznacza to, że nie jest możliwe uniwersalne wskazanie najlepszego algorytmu niezależnie od danych. Praktycy zajmujący się uczeniem maszynowym zdają sobie z tego doskonale sprawę i bardzo często ich praca polega na eksperymentalnym próbowaniu wielu algorytmów dla nowo otrzymanego zadania.

Pomimo tej swoistej trudności, istnieją pewne rezultaty matematyczne, które

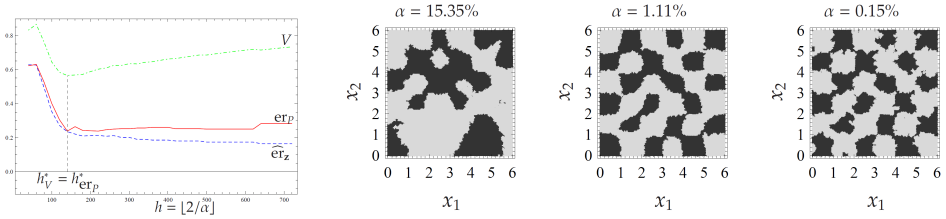
mówią, kiedy (przy jakich warunkach) i w jakim stopniu uczenie maszynowe ma szanse powieść się — tj. kiedy możemy spodziewać się, że w wyniku uczenia otrzymamy dobrze uogólniający dokładny model. A zatem kluczową pożądaną własnością jest wysoka ***zdolność do uogólniania*** (ang. *generalization capability*) maszyny uczącej się [Klę12]. Wspomniane rezultaty zostały sformułowane w ramach tzw. ***Statystycznej Teorii Uczenia*** (ang. *Statistical Learning Theory*) zapoczątkowanej przez Vladimira Vapnika [Vap95; Vap98; VC71] z wykorzystaniem techniki matematycznej ***PAC*** (ang. *Probably Approximately Correct*) pochodzącej od Valianta [Val84].

SLT i PAC zwracają szczególną uwagę na relację pomiędzy rozmiarem próby uczącej a złożonością przyjętego modelu, i na tej podstawie dostarczają nam ilościowych wyników[1] na temat zdolności do uogólniania otrzymywanych modeli. Złożoność modelu można mierzyć na wiele sposobów — np. poprzez: liczbę nastrajalnych parametrów, wymiar Vapnika-Chervonenkisa, liczby pokryciowe, złożoność Rademachera, i wiele innych. Oczywiście, modele o zbyt dużej złożoności w stosunku do rozmiaru próby uczącej, nazywane także przewymiarowanymi, mają tendencję do *przeuczania się* (ang. *overfitting*), czyli efektu przeciwnego do dobrego uogólniania. Model przeuczony charakteryzuje się bardzo małym błędem na danych uczących i istotnie większym błędem na danych testowych (nie widzianych podczas uczenia). A zatem interesującymi są ilościowe odpowiedzi na pytania: *jak dobrze uogólniamy?* lub równoważnie *jak mocno przeuczamy?*

Rys. 6.1 przedstawia przykład zagadnienia wyboru złożoności modelu i daje Czytelnikowi intuicyjny pogląd na treści omawiane w niniejszym rozdziale. Rysunek dotyczy znanego prostego algorytmu „najbliższych sąsiadów", za pomocą którego klasyfikowany jest wzorzec szachownicy. Tradycyjnie, pewna liczba k najbliższych sąsiadów decyduje o złożoności modelu w przypadku tego algorytmu. Im mniejsze k (bliższe 1), tym model bardziej złożony, a wynikowa granica decyzyjna bardziej „powyginana". Na przedstawionym rysunku zamiast k obserwowana jest równoważnie liczba $\alpha \in (0,1)$ jako złożoność modelu, reprezentująca procent najbliższych sąsiadów. Wykres po lewej stronie to przebieg procedury wyboru złożoności modelu, gdzie obserwowane są: błąd na próbie uczącej (niebieska krzywa), tzw. błąd prawdziwy (czerwona krzywa), i ograniczenie na tenże błąd oparte na tzw. wymiarze Vapnika-Chervonenkisa (zielona krzywa) — wynoszącym w tym przypadku $\lceil 2/\alpha \rceil$. Pojęcia błędu prawdziwego i wymiaru VC zostaną omówione w tym rozdziale. Kolejne rysunki pokazują trzy wybrane przykładowe modele: niewystarczająco złożony (dla $\alpha = 15.35\%$ najbliższych sąsiadów), odpowiednio dobrze złożony (dla $\alpha = 1.11\%$ najbliższych sąsiadów), zbyt złożony (dla $\alpha = 0.15\%$ najbliższych sąsiadów) — czyli przeuczony, dopasowujący się do

[1] zwykle w postaci probabilistycznych ograniczeń (nierówności)

szumów w danych uczących.



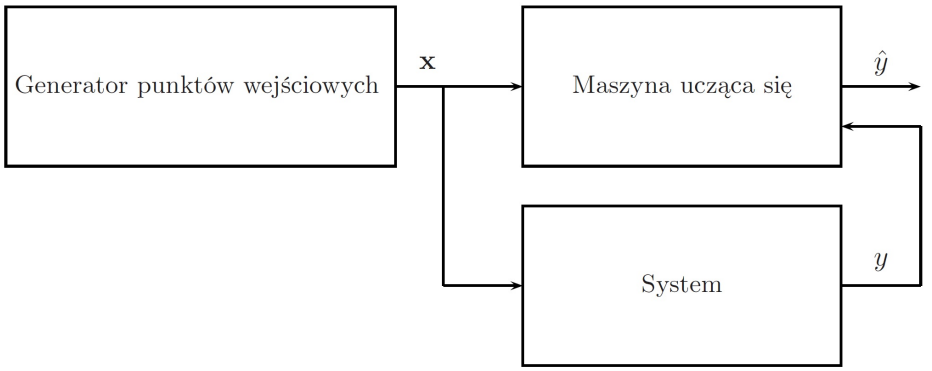
Rys. 6.1: Klasyfikacja wzorca „szachownica" z wykorzystaniem algorytmu najbliższych sąsiadów (źródło: (KK11)). Wykres po lewej stronie pokazuje przebieg procedury wyboru złożoności modelu (złożoność określona przez parametr α — procent najbliższych sąsiadów), gdzie obserwowane są: błąd na próbie (niebieska przerywana krzywa), błąd prawdziwy (czerwona krzywa), i ograniczenie na błąd oparte na wymiarze VC (zielona krzywa). Kolejne rysunki pokazują trzy wybrane modele: niewystarczająco złożony, odpowiednio dobrze złożony, zbyt złożony (przeuczony), (źródło: *opracowanie własne*).

Niniejszy rozdział omawia formalne matematyczne podstawy Statystycznej Teorii Uczenia (SLT). Treści rozdziału są trudne, ale zachęcamy do ich lektury, nawet jeżeli Czytelnik nie zdoła wykorzystać ich w przyszłości bezpośrednio. Naszym zdaniem treści te budują pewne wyobrażenie i wyczucie, na co praktyk powinien zwracać uwagę, rozwiązując zadanie uczenia maszynowego.

6.1 Ogólny scenariusz uczenia się z danych

Algorytmy uczenia maszynowego są w większości przypadków stosowane w tzw. *sytuacji obserwacyjnej* (ang. *observational setting*). Jest to sytuacja częsta w rzeczywistości. Jesteśmy biernymi obserwatorami pewnego zjawiska, odnotowujemy pochodzące z niego dane, natomiast nie znamy mechanizmu rządzącego tymże zjawiskiem. Mówiąc ściślej nie znamy łącznego rozkładu prawdopodobieństwa, według którego objawiają się obserwowane wielkości.

Zadanie uczenia się z danych polega na estymacji nieznanej zależności wejściowo-wyjściowej na podstawie skończonej liczby obserwacji. Ogólny scenariusz tego zadania obrazuje schemat pokazany na rys. 6.2. Zawiera on trzy podstawowe elementy: *generator* losowych punktów (wektorów) wejściowych, *system*, który zwraca wartości wyjściowe dla danych punktów wejściowych, oraz *maszynę uczącą się*, która obserwujac przykłady (czyli pary: punkt wejściowy i wartość wyjściowa) dokonuje estymacji nieznanego odwzorowania wejściowo–wyjściowego. Powyższe sformułowanie obejmuje dwa najważniejsze zadania uczenia nadzorowanego: klasyfikację i estymację funkcji regresji (lub inaczej aproksymację).



Rys. 6.2: Maszyna ucząca się na podstawie obserwacji systemu (źródło: *opracowanie własne* na podstawie (AB09; CM07; RM99)).

Generator generuje punkty losowe $\mathbf{x} \in \mathbb{R}^n$, $\mathbf{x} = (x_1, \ldots, x_n)$, wybierane niezależnie z rozkładu o pewnej *stałej* funkcji gęstości $p(\mathbf{x})$, która zwykle jest *nieznana* dla modelującego. Jak wspomniano wcześniej, w terminologii statystycznej taka sytuacja nazywana jest *obserwacyjną*. Oznacza to, że modelujący nie ma wpływu na to, jakie wartości wejściowe dostarczane sa do systemu, a sam generator można traktować jako część systemu. Przeciwieństwem sytuacji obserwacyjnej jest sytuacja nazywana *eksperymentem planowanym* lub *kontrolowanym*, w której modelujacy ma możliwość sam wymuśic określony i deterministyczny schemat próbkowania [CM07].

System dostarcza wartość wyjściową y dla każdego punktu $\mathbf{x}$ zgodnie ze stałym warunkowym rozkładem prawdopodobieństwa określonym przez prawdopodobieństwa $P(y|\mathbf{x})$ w przypadku gdy wartości y są dyskretne (zadanie klasyfikacji) lub przez gęstość $p(y|\mathbf{x})$ w przypadku gdy wartości y są ciągłe (zadanie estymacji regresji). Taki warunkowy rozkład jest również *nieznany*. Powyższy opis zawiera szczególny przypadek systemu deterministycznego, gdzie dla każdego ustalonego $\mathbf{x}$ system zwraca zawsze tę samą odpowiedź, ale także ogólniejszy przypadek, w którym dla tego samego $\mathbf{x}$ system z pewnym rozkładem prawdopodobieństwa zwraca różne odpowiedzi. Możemy tu myśleć albo o zadaniu aproksymacji i losowych odchyłkach wokół pewnej rzeczywistoliczbowej wartości średniej[2] lub też o zadaniu klasyfikacji i o losowych „przekłamaniach" typowej etykiety klasy dla danego $\mathbf{x}$. Systemy rzeczywiste rzadko kiedy mają wyjścia całkiem losowe, mają natomiast pewne nieznane lub nieuwzględnione wejścia. Efekt tych wejść na wyjście można rozumieć właśnie jako zmienną losową o pewnym rozkładzie prawdopodobieństwa

[2] ściślej mówiąc, o odchyłkach wokół funkcji regresji

[CM07; Ger99; Klę05].

Maszyna ucząca się jest obiektem wyposażonym w pewien z góry przyjęty *zbiór funkcji* $F = \{f\}$ (nazywanych czasem także hipotezami) oraz *algorytm uczący*, który na podstawie dostarczonych przykładów danych (obserwacji) potrafi wybrać jedną funkcję ze zbioru F jako model systemu. Tym samym, po wybraniu takiej jednej funkcji, maszyna będzie w stanie zwracać swoje odpowiedzi (rozpoznania, przewidywania) dla nowo przychodzących punktów $\mathbf{x}$, tj. zwracać $\hat{y} = f(\mathbf{x})$.

Wyróżnia się dwa ważne typy zbiorów funkcji, którymi posługują się maszyny uczące się: *liniowe* ze wzgledu na parametry i *nieliniowe* ze wzgledu na parametry. Należy zaakcentować tu fakt, że liniowość (czy też nieliniowość) dotyczy właśnie parametrów, nie zaś zmiennych wejściowych. Na przyklad zbiór zawierający funkcje trygonometryczne postaci

$$f(x;\mathbf{w},\mathbf{v}) = w_0 + \sum_{k=1}^{N} \left(v_k \sin(kx) + w_k \cos(kx)\right), \tag{6.1}$$

jest zbiorem liniowym ze względu na parametry, podobnie jak chociażby zbiór funkcji wielomianowych postaci

$$f(x;\mathbf{w}) = \sum_{k=0}^{N} w_k x^k. \tag{6.2}$$

Natomiast zbiór zawierająca funkcje typu

$$f(\mathbf{x};\mathbf{w},\mathbf{V}) = w_0 + \sum_{k=1}^{N} w_k \phi\left(v_{k0} + \sum_{j=1}^{n} v_{kj} x_j\right), \tag{6.3}$$

gdzie ϕ jest pewną funkcją nieliniową (np. sigmoidalną), jest tym samym zbiorem nielinioweym, jako że parametry $\mathbf{V} = \{v_{kj}\}$ są pod dzialaniem funkcji ϕ. Taki zbiór funkcji służy m.in. do reprezentowania sieci neuronowych z jedna nieliniową warstwą ukrytą [Klę05], o których była mowa w punkcie 4.2.

Rozróżnienie na liniowe i nieliniowe zbiory funkcji jest ważne, dlatego że w procesie strojenia modelu przekłada się ono na rozwiązywanie odpowiednio liniowego i nieliniowego zadania optymalizacji [CM07; Ger99; Klę05].

Podsumowując, w ujęciu teorii SLT (i techniki PAC) na maszynę uczącą się patrzymy jak na parę: (1) zbiór funkcji matematycznych, który ma ona do dyspozycji oraz (2) algorytm uczący, który mówi, w jaki sposób należy wybrać jedną funkcję z tego zbioru. Zadanie uczenia się z danych dobrze jest wówczas postawić jako problem, który należy rozwiązać z zadaną z góry (ε,δ)-*precyzją*. Oznacza to, że algorytm uczący będzie wybierał funkcję, która popełnia błąd prawdziwy (to pojęcie zdefiniujemy już za chwilę) nie gorszy niż o ε od błędu najlepszej

funkcji możliwej do osiągnięcia w przyjętym zbiorze i fakt ten będzie miał miejsce z prawdopodobieństwem przynajmniej $1-\delta$.

W kolejnej sekcji uściślamy naszkicowane dotychczas pojęcia. Przyjęta notacja i nazewnictwo są zgodne z powszechnie przyjętymi w pracach na temat SLT (patrz np. [AB09; Vap95; Vap98]). Duża część spośród następujących treści jest powtórzona za pracą [Klę12].

6.2 Notacja i pojęcia podstawowe

Niech

$$\mathbf{z} = \{\mathbf{z}_1, \mathbf{z}_2, \ldots, \mathbf{z}_m\} = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_m, y_m)\}. \tag{6.4}$$

oznacza ***próbę*** (ang. *sample*) o rozmiarze m, tj. zbiór par $(\mathbf{x}, y)$ czerpanych z pewnego *nieznanego*, ale *stałego* łącznego rozkładu prawdopodobieństwa P. Rozkład P reprezentuje dane zjawisko, które jest przedmiotem uczenia. Pojedyncze punkty danych czerpane są w sposób i.i.d. (ang. *independent, identically distributed*), czyli niezależnie i przy zachowaniu stałego rozkładu, i tym samym możemy myśleć o produktowym rozkładzie P^m, z którego pochodzi cała próba. Jeżeli chodzi o dziedziny, niech w ogólności $\mathbf{x} \in \mathbf{X} \subset \mathbb{R}^n$ oraz $y \in Y$, gdzie w zależności od rodzaju zadania dziedzinę Y będzie stanowił pewien zbiór skończony (zadanie klasyfikacji), lub zbiór $\mathbb{R}$ (zadanie estymacji funkcji regresji) [Klę12]. Dla uproszczenia pominiemy w dalszych rozważaniach zadania estymacji gęstości i klasteryzacji, które są zadaniami uczenia nienadzorowanego (ang. *unsupervised learning*)[3].

Niech

$$F = \{f\}, \tag{6.5}$$

gdzie $f\colon \mathbf{X} \to Y$, oznacza ***zbiór funkcji***, który maszyna ucząca się ma do dyspozycji. Za pomocą pewnej wybranej funkcji z tego zbioru będziemy chcieli przybliżać badane zjawisko.

Pojęcie ***zdolności do uogólniania*** dla pewnej ustalonej funkcji f można utożsamiać z liczbową wartością ***błędu prawdziwego***[4] (ang. *true error*) popełnianego przez tę funkcję [Klę12]. Chodzi tu o błąd policzony w sposób dokładny jako wartość oczekiwana względem rozkładu P. Dla zadania klasyfikacji błąd prawdziwy definiujemy jako:

$$\mathrm{er}_P(f) = \int\limits_{\mathbf{x}\in\mathbf{X}} \sum_{y\in Y} [f(\mathbf{x}) \neq y] \underbrace{p(\mathbf{x})P(y|\mathbf{x})\mathbf{dx}}_{dP(\mathbf{x},y)}, \tag{6.6}$$

[3] tj. nie uwzględniają wartości y pochodzących z systemu lub też wartości te nie są obserwowane

[4] Wielkość ta bywa też nazywana *ryzykiem prawdziwym* (ang. *true risk*) np. u Vapnika [Vap95; Vap98].

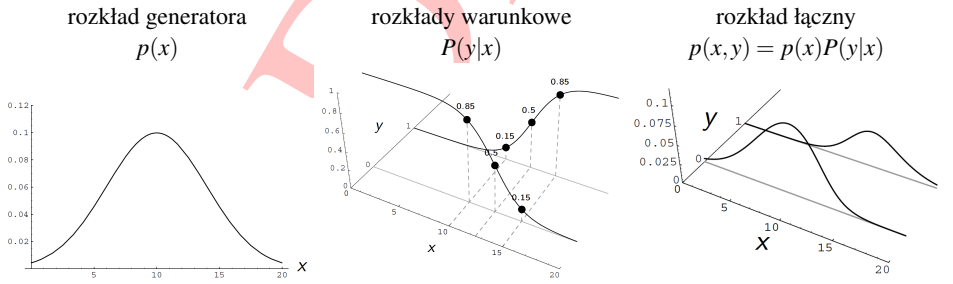
gdzie notacja $[\cdot]$ jest funkcją wskaźnikową, przyjmującą 1 gdy zdanie będące argumentem jest prawdziwe i 0 w przeciwnym razie. Jak można zauważyć, liczbowy sens er_P to *prawdopodobieństwo błędnego sklasyfikowania* losowej pary $(\mathbf{x}, y)$ zaczerpniętej z P. Dla zadania estymacji funkcji regresji błąd prawdziwy definiujemy zwykle[5] jako:

$$\mathrm{er}_P(f) = \int_{\mathbf{x}\in\mathbf{X}} \int_{y\in Y} (f(\mathbf{x}) - y)^2 \underbrace{p(\mathbf{x})p(y|\mathbf{x})dy\mathbf{dx}}_{dP(\mathbf{x},y)}. \tag{6.7}$$

Liczbowy sens er_P to w tym przypadku to *oczekiwany kwadrat odchyłki* pomiędzy $f(\mathbf{x})$ a y. Warto dodać, że sama *funkcja regresji*, którą staramy się przybliżać za pomocą f, jest w każdym punkcie $\mathbf{x}$ określona jako $r(\mathbf{x}) = \int_{y\in Y} p(y|\mathbf{x})dy$, czyli jako wartość oczekiwana (średnia) z rozkładu warunkowego.

Funkcje podcałkowe w powyższych definicjach, tj. $[f(\mathbf{x}) \neq y]$ oraz $(f(\mathbf{x}) - y)^2$ odpowiednio dla klasyfikacji i estymacji regresji, są w nomenklaturze SLT określane mianem *funkcji straty* (ang. *loss functions*).

Dla lepszego zrozumienia pojęć rozkładu łącznego P oraz błędu prawdziwego er_P rozważmy następujący konkretny przykład problemu klasyfikacji z jedną zmienną wejściową[6] x oraz dwiema klasami $y \in \{0, 1\}$. Rys. 6.3 obrazuje rozkłady składowe (brzegowe) oraz łączny prawdopodobieństwa definiujące ten problem. Pierwszy z wykresów przedstawia rozkład generatora określony przez gęstość $p(x)$,



Rys. 6.3: Przykład problemu klasyfikacji z jedną zmienną wejściową. Problem jest zdefiniowany przez łączny rozkład prawdopodobieństwa (źródło: *opracowanie własne*).

czyli rozkład, wg którego objawiają się punkty x. Wykres środkowy reprezentuje warunkowe rozkłady prawdopodobieństwa $P(y|x)$, zgodnie z którymi objawiają

[5] W zadaniu estymacji funkcji regresji bardzo rzadko bywają używane inne funkcje niż kwadratowa funkcja błędu.

[6] Używamy tu celowo czcionki niepogrubionej, ponieważ x będzie skalarem (nie zaś wektorem cech).

się wartości y dla każdego x. Innymi słowy mamy tu do czynienia z nieskończenie wieloma rozkładami[7] dwupunktowymi — każdy z nich możemy odczytać ustalając odciętą x. Np. dla $x = 10$ mamy rozkład $P(y = 0|x) = 0.85$, $P(y = 1|x) = 0.15$. Można zatem zauważyć, że problem nie jest deterministyczny, tzn. dla tej samej wartości x system może odpowiedzieć różnymi wartościami klasy y, przy czym dla małych x klasa $y = 0$ jest bardziej prawdopodobna, zaś dla dużych x bardziej prawdopodobną jest klasa $y = 1$. Ten niedeterminizm może być tłumaczony np. przez fakt, że obserwujemy tylko jedną zmienną wejściową (w ogólności zazwyczaj jesteśmy w stanie „zmniejszyć" niedeterminizm obserwując lub mierząc więcej istotnych zmiennych wejściowych). Można także zauważyć, że $x = 12.5$ wydaje się być najlepszą wartością progową do rozróżniania pomiędzy klasami tj. do podejmowania decyzji. Niemniej, przy ostatecznym wyborze takiego progu należy także uwzględnić rozkład $p(x)$, a w naszym przykładzie rozkład ten jest wyśrodkowany wokół wartości 10. Wykres po prawej stronie rys. 6.3 pokazuje rozkład łączny nad parami (x,y), który jest określony iloczyn $p(x)P(y|x)$. To właśnie rozkład łączny można utożsamiać z problemem uczącym. Innymi słowy rozkład łączny (przy ustalonej przestrzeni cech) stanowi *pełną* informację o problemie. Poniżej podane są wzory rozkładów przyjętych na potrzeby omawianego przykładu:

$$p(x) = \frac{1}{\sqrt{2\pi}4.0} e^{-\frac{(x-10.0)^2}{2\cdot 4.0^2}},$$
$$P(y=0|x) = \frac{1}{1+e^{0.75(x-12.5)}}, \quad P(y=1|x) = 1 - \frac{1}{1+e^{0.75(x-12.5)}}. \qquad (6.8)$$

Dla powyższego przykładu rozważmy teraz maszynę uczącą się, która ma do dyspozycji następujący zbiór funkcji $F = \{f(x;\omega)\}_{\omega\in\mathbb{R}}$, gdzie

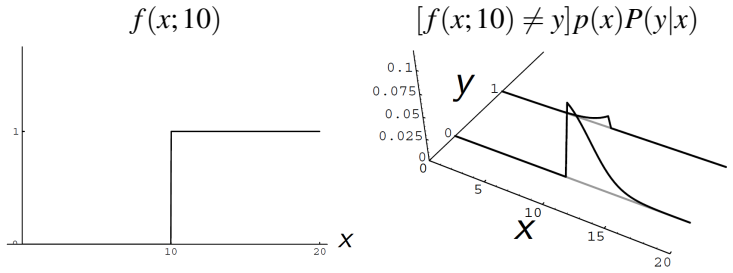
$$f(x;\omega) = \begin{cases} 1, & \text{dla } x > \omega; \\ 0, & \text{dla } x \leqslant \omega. \end{cases}$$

Parametr ω można traktować jak indeks konkretnej funkcji w tym zbiorze i stanowi on jednocześnie próg decyzyjny. Tego typu funkcje bywają w uczeniu maszynowym określane nazwą „decision stumps" (przy czym są one zwyczajowo wyposażane w jeszcze jeden parametr dedycujący o kierunku decyzji).

Przypuśćmy teraz, że chcielibyśmy ocenić, na ile dobre jako klasyfikatory byłyby na przykład funkcje $f(x;10)$ i $f(x;13)$, czyli funkcje z progami decyzyjnymi odpowiednio w punktach 10 i 13. Innymi słowy chcemy obliczyć i porównać błędy prawdziwe tych funkcji.

Rys. 6.4 przedstawia wykres funkcji $f(x;10)$ oraz jej funkcji straty ważonej rozkładem łącznym. To właśnie ta ważona funkcja straty będzie całkowana

[7] continuum rozkładów

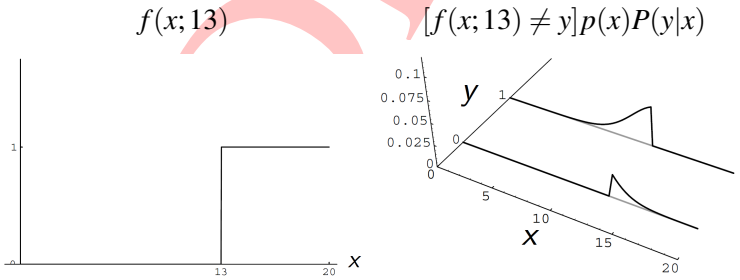


Rys. 6.4: Wykres funkcji $f(x;10)$ oraz jej funkcji straty ważonej łącznym rozkładem prawdopodobieństwa (źródło: *opracowanie własne*).

w celu obliczenia błędu prawdziwego. Używając dowolnego środowiska do obliczeń numerycznych można przekonać się, że błąd prawdziwy er_P w tym przypadku wynosi w przybliżeniu

$$\int_{-\infty}^{\infty} \sum_{y\in\{0,1\}} [f(x;10) \neq y]p(x)P(y|x)\,dx \approx 0.239671. \tag{6.9}$$

Postępując analogicznie dla funkcji $f(x;13)$ możemy wyznaczyć jej ważoną funkcję straty (patrz rys. 6.5) i tym samym obliczyć poprzez całkowanie jej



Rys. 6.5: Wykres funkcji $f(x;13)$ oraz jej funkcji straty ważonej łącznym rozkładem prawdopodobieństwa (źródło: *opracowanie własne*).

błąd prawdziwy:

$$\int_{-\infty}^{\infty} \sum_{y\in\{0,1\}} [f(x;10) \neq y]p(x)P(y|x)\,dx \approx 0.143819. \tag{6.10}$$

A zatem błąd prawdziwy popełniany przez funkcję $f(x;13)$ jest mniejszy (niż błąd prawdziwy funkcji $f(x;10)$) i to ją należałoby wybrać jako klasyfikator, gdyby wybór ograniczony był tylko do dwóch wspomnianych funkcji.

W powyższym przykładzie obliczenie błędów prawdziwych było możliwe tylko dzięki temu, że łączny rozkład prawdopodobieństwa był znany w sposób jawny, dany poprzez wzory (6.8). Należy mocno podekreślić, że błąd prawdziwy *nie* jest możliwy do obliczenia w spotykanych w praktyce typowych zadaniach uczenia maszynowego (z wyjątkiem eksperymentów kontrolowanych), ponieważ rozkład łączny nie jest znany, a do naszej dyspozycji jest tylko skończona próba pochodząca z tego rozkładu, patrz zapis (6.4). Interesującym natomiast jest to, że w ramach teorii SLT istnieją różne techniki pozwalające na szacowanie nieznanej wartości błędu prawdziwego.

Podstawową wielkością, która jest wyliczana i pojawia się w każdym praktycznym eksperymencie, jest ***błąd na próbie***[8] (ang. *sample error*), a mówiąc pełniej jest to błąd na próbie uczącej. Dla zadania klasyfikacji błąd na próbie $\widehat{\mathrm{er}}_{\mathbf{z}}$ obliczamy jako

$$\widehat{\mathrm{er}}_{\mathbf{z}}(f) = \frac{1}{m}\sum_{i=1}^{m}[f(\mathbf{x}_i) \neq y_i], \tag{6.11}$$

natomiast dla zadania estymacji funkcji regresji jako

$$\widehat{\mathrm{er}}_{\mathbf{z}}(f) = \frac{1}{m}\sum_{i=1}^{m}(f(\mathbf{x}_i) - y_i)^2. \tag{6.12}$$

Liczbowy sens (6.11) i (6.12) to odpowiednio *częstość błędnej klasyfikacji* oraz *średni kwadrat odchyłki* [Klę12].

Algorytm uczący L wybiera ze zbioru F jedną funkcję $\widehat{f}$ mając na uwadze zaobserwowane dane, i starając się, aby wybrana funkcja minimalizowała błąd na próbie, tj. aby:

$$\widehat{f} = \arg\inf_{f\in F} \widehat{\mathrm{er}}_{\mathbf{z}}(f). \tag{6.13}$$

Takie postępowanie nazywane jest w literaturze regułą indukcyjną SAE (ang. *sample error minimization*) lub ERM (ang. *empirical error minimization*)[9]. Innymi słowy na sam algorytm uczący możemy patrzeć jak na następujące odwzorowanie

$$L\colon \bigcup_{m=1}^{\infty}(\mathbf{X}\times\mathbf{Y})^m \to F, \tag{6.14}$$

które dla danej próby $\mathbf{z}$ wskazuje określoną hipotezę $L(\mathbf{z}) = \widehat{f}$ wybraną ze zbioru F. Jawne rozróżnienie pomiędzy zbiorem funkcji a algorytmem uczącym jest

[8] Bywa też nazywany *ryzykiem empirycznym* (ang. *empirical risk*).

[9] O ile arg inf istnieje. Jeżeli F jest zbiorem *zwartym*, to wówczas funkcjonały $\widehat{\mathrm{er}}_{\mathbf{z}}$ i er_P zdefiniowane nad zbiorem zwartym osiągają swoje kresy na mocy twierdzenia Weierstrassa.

przydatne. Możemy pomyśleć np. o sieci neuronowej, gdzie funkcjami w zbiorze F są kombinacje lub złożenia sigmoid, i uczyć tę sieć różnymi algorytmami L: klasycznym algorytmem backpropagation, algorytmem RPROP, algorytmem Adam, metodą największej wiarygodności itd. Inny przykład — zbiór funkcji mogą stanowić wielomiany, które można uczyć metodą najmniejszych kwadratów: bez regularyzacji na współczynniki, z regularyzacją ℓ_2, z regularyzacją ℓ_1 itp. [Klę12]

Jak można zauważyć, błąd na próbie jest w zapisie pokrewny do błędu prawdziwego. Odpowiednie całki zastąpiono sumami. Jednakże, jak wiadomo, nie należy sądzić, że dla wybranej funkcji $\widehat{f}$ wartość jej błędu na próbie (zdolność do odtwarzania) to dyskretny odpowiednik lub oszacowanie dla jej błędu prawdziwego (zdolność do uogólniania). W większości przypadków błędy na próbie są mniejsze niż błędy prawdziwe, jako że są one (błędy na próbie) osiągnięte poprzez wybór funkcji dobrze dopasowanej do konkretnych danych uczących, a tym samym szumów obecnych w tych danych. Mówiąc inaczej, można łatwo wskazać funkcję, dla której błąd na próbie jest bliski zeru lub nawet zero, a jednocześnie funkcja ta słabo uogólnia tj. ma duży błąd prawdziwy. Mamy wówczas do czynienia z przeuczeniem.

Z drugiej strony warto nadmienić, że częstą praktyką jest dzielenie całości danych na próbę uczącą i *próbę testową*. Jeżeli ma to miejsce i w próbie testowej znajdują się przykłady, których algorytm uczący nie widział podczas uczenia, to wówczas błąd obliczony na próbie testowej (odpowiednio dużej) jest faktycznie przybliżeniem błędu prawdziwego. Uściślając, jeżeli przez $\mathbf{z}' = \{\mathbf{z}'_1, \ldots, \mathbf{z}'_{m'}\} = \{(\mathbf{x}'_1, y'_1), \ldots, (\mathbf{x}'_{m'}, y'_{m'})\}$ oznaczymy pewną wydzieloną „na bok" próbę testową, gdzie m' oznacza jej rozmiar, to wówczas prawdą jest, że $\widehat{\mathrm{er}}_{\mathbf{z}'}(f) \approx \mathrm{er}_P(f)$ dla dowolnej funkcji f. Co więcej, w granicy dla $m' \to \infty$ znak przybliżenia w powyższym stwierdzeniu należy zastąpić równością.

Niech f^* oznacza najlepszą funkcję w zbiorze F, taką że:

$$f^* = \arg\inf_{f \in F} \mathrm{er}_P(f). \tag{6.15}$$

Oczywiście chcielibyśmy, żeby funkcja $\widehat{f}$ wybrana przez algorytm uczący miała błąd prawdziwy $\mathrm{er}_P(\widehat{f})$ jak najbliższy do $\mathrm{er}_P(f^*)$.

Oprócz rozważania zbioru F dobrze jest w pewnych kontekstach patrzeć równolegle na ***zbiór funkcji straty*** (ang. *loss functions*). Chodzi tu o zbiór:

$$l_F = \{l_f \colon \ f \in F\}, \tag{6.16}$$

gdzie dla zadania klasyfikacji mamy funkcje $l_f(\mathbf{z}) = l_f((\mathbf{x}, y)) = [f(\mathbf{x}) \neq y]$ realizujące odwzorowanie zero-jedynkowe $l_f \colon \ \mathbf{X} \times Y \to \{0, 1\}$, a dla zadania estymacji regresji mamy funkcje $l_f(\mathbf{z}) = (f(\mathbf{x}) - y)^2$ realizujące odwzorowanie $l_f \colon \ \mathbf{X} \times Y \to \mathbb{R}$.

Zbiory F i l_F są ze sobą bezpośrednio skojarzone, a dla szczególnego przypadku klasyfikacji binarnej pomiędzy F i l_F istnieje odpowiedniość $1:1$. Warto dodatkowo zauważyć, że dla zadania klasyfikacji, niezależnie od liczby klas w problemie (tj. niezależnie od liczności przeciwdziedziny, do której odwzorowują funkcje $f\colon \mathbf{X} \to Y$) funkcje l_f są zawsze funkcjami zero-jedynkowymi. Ten fakt ma znaczenie przy definicji wymiaru Vapnika-Chervonenkisa, o którym później.

6.3 Zbieżność jednostajna i pojęcia złożoności maszyn uczących się

6.3.1 Zbieżność jednostajna dla skończonych zbiorów funkcji zero-jedynkowych

Jednym z podstawowych celów teorii SLT jest badanie tempa ***jednostajnej zbieżności w prawdopodobieństwie*** (ang. *uniform convergence in probability*) błędów na próbie do błędów prawdziwych, gdyby generować ciąg takich wyników wraz z podnoszeniem rozmiaru m próby uczącej. Jednostajność oznacza, że interesuje nas najbardziej pesymistyczny przypadek, który może mieć miejsce — to znaczy, pytamy o prawdopodobieństwo takiego zdarzenia, że największa odchyłka pomiędzy błędem prawdziwym a błędem na próbie (która ma miejsce dla pewnej funkcji f w zbiorze F) przekracza pewien zadany próg ε, tj.: $\sup_{f\in F} |\mathrm{er}_P(f) - \widehat{\mathrm{er}}_{\mathbf{z}}(f)| > \varepsilon$. Należy mieć świadomość, że tego typu zdarzenie może przy pewnych złośliwych okolicznościach w szczególności zachodzić właśnie dla funkcji $\hat{f}$, czyli tej, którą wybiera się poprzez minimalizację błędu na próbie.

Przydatnym narzędziem do twierdzeń o jednostajnej zbieżności jest nierówność Chernoffa. Opisuje ona związek pomiędzy prawdopodobieństwem p pewnego zdarzenia[10], a jego częstością ν_m zaobserwowaną na próbie o rozmiarze m:

$$P_m(|p - \nu_m| > \varepsilon) \leqslant 2e^{-2\varepsilon^2 m}, \tag{6.17}$$

gdzie prawdopodobieństwo P_m jest wyliczane względem przestrzeni wszystkich prób o rozmiarze m. Jak widać prawdopodobieństwo odchyłki większej niż ε maleje w tempie wykładniczym ze względu na rozmiar próby. Istnieją także wersje jednostronne nierówności Chernoffa:

$$P_m(p - \nu_m > \varepsilon) \leqslant e^{-2\varepsilon^2 m}, \tag{6.18}$$

$$P_m(\nu_m - p > \varepsilon) \leqslant e^{-2\varepsilon^2 m}. \tag{6.19}$$

Rozważmy najprostszy przypadek *skończonego* zbioru funkcji $F = \{f_1, \ldots, f_N\}$ użytego do uczenia. Znany jest następujący elementarny rezultat [Vap98; VC71] na

[10] Uwaga: nie należy w tym kontekście odczytywać oznaczenia p jako funkcji gęstości.

temat jednostajnej zbieżności:

$$P_m\left(\sup_{f\in F}(\mathrm{er}_P(f)-\widehat{\mathrm{er}}_{\mathbf{z}}(f))>\varepsilon\right)\leqslant\sum_{k=1}^{N}P_m\left(\mathrm{er}_P(f_k)-\widehat{\mathrm{er}}_{\mathbf{z}}(f_k)>\varepsilon\right)\leqslant N\cdot e^{-2\varepsilon^2 m}, \tag{6.20}$$

gdzie ostatnie przejście wynika z faktu, że dla każdej ustalonej funkcji f_k zachodzi nierówność Chernoffa (6.18)[11]. Przypisując do prawej strony nierówności (6.20) pewne małe prawdopodobieństwo δ i rozwiązując ze względu na ε, powyższy rezultat można równoważnie wyrazić w formie ***ograniczenia na błąd prawdziwy***[12]:

$$\mathrm{er}_P(f_k)\leqslant\widehat{\mathrm{er}}_{\mathbf{z}}(f_k)+\sqrt{\frac{\ln N-\ln\delta}{2m}}, \tag{6.21}$$

które zachodzi z prawdopodobieństwem przynajmniej $1-\delta$ dla *każdej* funkcji f_k w zbiorze F. W szczególności zachodzi też więc dla $\widehat{f}$. Idąc dalej można łatwo pokazać[13], że z prawdopodobieństwem przynajmniej $1-2\delta$:

$$\mathrm{er}_P(\widehat{f}\,)-\mathrm{er}_P(f^*)\leqslant\sqrt{\frac{\ln N-\ln\delta}{2m}}+\sqrt{\frac{-\ln\delta}{2m}}, \tag{6.22}$$

co stanowi ograniczenie na różnicę pomiędzy błędem prawdziwym wybranej funkcji $\widehat{f}$ a błędem prawdziwym najlepszej możliwej funkcji f^* w przyjętym F. Należy przypomnieć, że oba te błędy prawdziwe są w praktyce nieznane, a mimo to — co ciekawe — podanie ograniczenia jest możliwe [Klę12].

6.3.2 Złożoność próbkowa

Kolejnym ważnym pojęciem jest ***złożoność próbkowa*** (ang. *sample complexity*) oznaczana jako $m_L(\varepsilon,\delta)$. Złożoność próbkowa to minimalny rozmiar próby wystarczający na uczenie algorytmem L z zadaną (ε,δ)-precyzją dla danego problemu. Ograniczenia na złożoność próbkową otrzymuje się bezpośrednio z ograniczeń w stylu nierówności (6.22)[14]. I tak dla uproszczonego przypadku skończonego zbioru funkcji złożoność próbkowa jest ograniczona następująco:

$$m_L(\varepsilon,\delta)\leqslant\frac{1}{2\varepsilon^2}\left(\sqrt{\ln N-\ln(\delta)}+\sqrt{-\ln(\delta)}\right)^2. \tag{6.23}$$

[11] Która stosuje się, ponieważ dla klasyfikacji wielkości er_P i $\widehat{\mathrm{er}}_{\mathbf{z}}$ oznaczają odpowiednio prawdopodobieństwo i częstość błędnego sklasyfikowania.

[12] Użyto jednostronnej wersji nierówności Chernoffa, ponieważ interesuje nas ograniczenie z góry.

[13] Wystarczy wykorzystać dwa fakty: (1) z definicji $\widehat{f}$ mamy $\widehat{\mathrm{er}}_{\mathbf{z}}(f^*)\geqslant\widehat{\mathrm{er}}_{\mathbf{z}}(\widehat{f}\,)$, oraz (2) dla f^* zachodzi bezpośrednio nierówność Chernoffa.

[14] Należy przypisać ε do lewej strony nierówności i rozwiązać ze względu na m.

W omówionym przypadku wielkością reprezentującą złożoność (bogatość) zbioru F była liczba N — liczba funkcji w zbiorze. Oczywiście nie jest to przypadek praktyczny, i tak naprawdę w praktyce interesuje nas uczenie w oparciu o nieskończone zbiory funkcji (continuum funkcji). Dla tych zbiorów, ograniczenia na błąd prawdziwy i złożoność próbkową są budowane w oparciu o inne pojęcia złożoności zbioru F (nazywane także: bogatością lub pojemnością, ang. *function set capacity*). Mówiąc skrótowo należy zastąpić pojawiający się $\ln N$ pewnym odpowiednikiem właściwym dla nieskończonego zbioru F. I tak dla nieskończonych zbiorów funkcji zero-jedynkowych (klasyfikacja) jest to zwykle logarytm z tzw. *funkcji wzrostu* (ang. *growth function*), a dla nieskończonych zbiorów funkcji rzeczywistych (estymacja regresji) jest to zwykle logarytm z *liczby pokryciowej* (ang. *covering number*).

6.3.3 Zbieżność jednostajna dla nieskończonych zbiorów funkcji zero-jedynkowych

Niech F oznacza nieskończony zbiór funkcji. Dla ustalonej próby $\mathbf{z}_1,\ldots,\mathbf{z}_m$ niech $(l_F)_{|\mathbf{z}_1,\ldots,\mathbf{z}_m}$ oznacza zbiór funkcji straty rozróżnialnych nad tą próbą (lub inaczej: zbiór funkcji odciętych do próby), tj.:

$$(l_F)_{|\mathbf{z}_1,\ldots,\mathbf{z}_m} = \Big\{ \big(l_f(\mathbf{z}_1),\ldots,l_f(\mathbf{z}_m)\big) :\ f \in F \Big\}. \tag{6.24}$$

Oczywiście $\#(l_F)_{|\mathbf{z}_1,\ldots,\mathbf{z}_m} \leqslant 2^m$.

Ważnym pojęciem w tym kontekście jest ***roztrzaskiwanie*** (ang. *shattering*).

Definicja 6.3.1 Mówimy, że zbiór funkcji zero-jedynkowych *roztrzaskuje* próbę $\mathbf{z}_1,\ldots,\mathbf{z}_m$, jeżeli w tym zbiorze istnieje 2^m funkcji rozróżnialnych nad próbą.

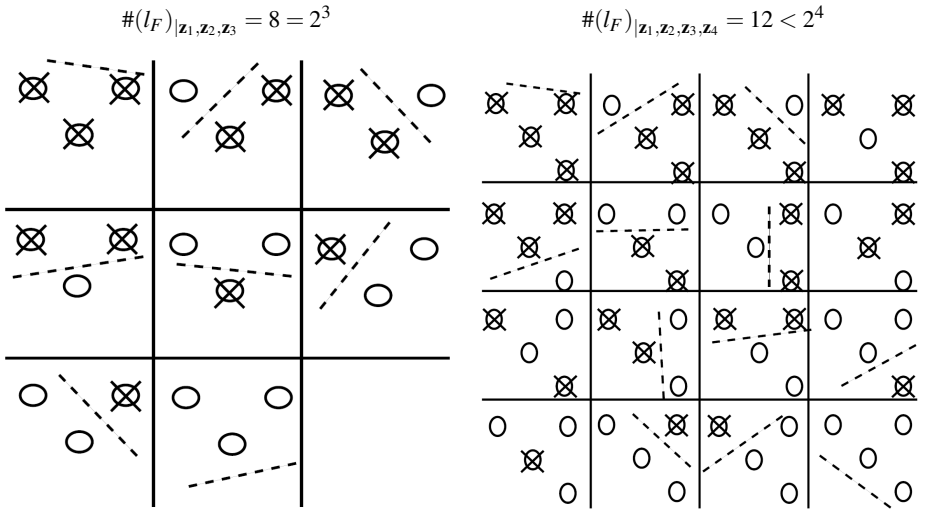
Inaczej mówiąc, oznacza to, że można zrealizować wszystkie dychotomie próby za pomocą funkcji z tego zbioru.

Idąc dalej, za pomocą pojęcia roztrzaskiwania można w poniższy sposób zdefiniować ***wymiar Vapnika-Chervonenkisa*** [VC71].

Definicja 6.3.2 Mówimy, że zbiór l_F funkcji zero-jedynkowych ma wymiar Vapnika-Chervonenkisa równy h, piszemy VC-dim $(l_F) = h$, wtedy i tylko wtedy, gdy istnieje próba o rozmiarze h roztrzaskiwana przez l_F i nie istnieje żadna taka próba o rozmiarze $h+1$. Jeżeli dla każdego $h > 0$ istnieje pewna próba roztrzaskiwana, to VC-dim $(l_F) = \infty$.

Rys. 6.6 stanowi przykładową ilustrację pokazującą, w jaki sposób zliczane są rozróżnialne funkcje (i tym samym dychotomie) nad próbą uczącą, w przypadku gdy rozważamy linie proste jako granice decyzyjne (co ma miejsce np. w perceptronie

prostym). W związku z tym, że nie istnieje próba 4-elementowa, która byłaby roztrzaskiwana przez taki zbiór funkcji (obejmujący różne ustawienia jednej linii prostej), to wymiar VC jest równy w tym przypadku 3.



Rys. 6.6: Liczba rozróżnialnych funkcji straty nad próbą 3-elementową i 4-elementową przy dyskryminacji za pomocą jednej linii prostej.

Znane są pewne zbiory funkcji, dla których dokładna wartość wymiaru VC została ustalona poprzez odpowiedni dowód kombinatoryczny. Oto niektóre przykłady. Dla płaszczyzn w $\mathbb{R}^n$ (hiperpłaszczyzn), które mogą być funkcjami bazowymi np. dla sieci perceptronowych, wymiar VC wynosi $n+1$ [Vap98]. Dla kostek w $\mathbb{R}^n$ wymiar VC wynosi $2n$ [CM07]. Dla kul w $\mathbb{R}^n$, które mogą być bazami dla radialnych sieci neuronowych, wymiar VC wynosi $n+1$ [CM07]. Dla wielomianów zdefiniowanych nad $\mathbb{R}^n$ stopnia co najwyżej s, wymiar VC wynosi $\binom{s+n}{n}$, patrz np. [AB09]. Jeżeli chodzi o liniowe kombinacje baz, to wymiar VC można zwykle ograniczyć z góry przez iloczyn liczby baz i wymiaru VC pojedynczej bazy [AB09, str. 154], ale to stwierdzenie wymaga zwykle ostrożnej analizy. Warto dodać jednocześnie, że istnieją zbiory funkcji, dla których nie udało się jeszcze określić (dowieść) wymiaru VC, a mimo to zbiory te są wykorzystywane w uczeniu [AB09; Klę12].

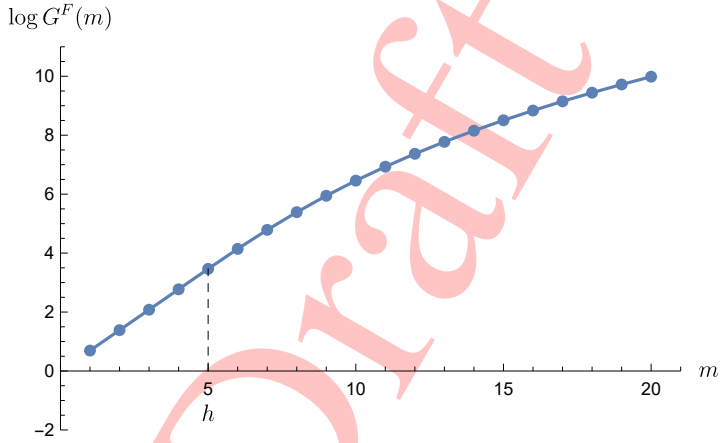
Alternatywnie, wymiar VC można definiować w oparciu o ***funkcję wzrostu*** (ang. *growth function*), która podaje największą liczbę rozróżnialnych funkcji dla danego rozmiaru próby:

$$G^F(m) = \sup_{\mathbf{z}\in(\mathbf{X}\times Y)^m} \#(l_F)_{|\mathbf{z}}. \tag{6.25}$$

Wymiar VC, to największy argument funkcji wzrostu, powyżej którego przestaje ona narastać wykładniczo. Jednym z istotnych rezultatów w tym kontekście jest lemat Sauera [CRS94; Sau72; Ste78], który mówi, że jeżeli VC-dim$(l_F) = h$, to:

$$G^F(m) \leqslant \sum_{i=0}^{h} \binom{m}{h} < \left(\frac{me}{h}\right)^h. \tag{6.26}$$

Na rys. 6.7 przedstawiono przykładowy wykres zlogarytmowanej funkcji wzrostu przy ustaleniu wymiaru VC na wartość $h = 5$.



Rys. 6.7: Przykładowy wykres logarytmu z funkcji wzrostu przy $h = 5$ (źródło: *opracowanie własne*).

Poniższe twierdzenie o jednostajnej zbieżności (z wykorzystaniem lematu Sauera) sformułowali oryginalnie Vapnik i Chervonenkis [VC71], patrz także [AB09].

Twierdzenie 6.3.1 Niech F będzie nieskończonym zbiorem funkcji zero-jedynkowych o funkcji wzrostu $G^F(m)$ i VC-dim$(F) = h$. Wówczas:

$$P_m\left(\sup_{f\in F} |\mathrm{er}_P(f) - \widehat{\mathrm{er}}_{\mathbf{z}}(f)| \geqslant \varepsilon\right) \leqslant 4G^F(2m)e^{-m\varepsilon^2/8} \tag{6.27}$$

$$\leqslant 4e^{h\left(1+\ln\frac{2m}{h}\right)-m\varepsilon^2/8}. \tag{6.28}$$

Analogicznie do wzoru (6.21), powyższy rezultat można zapisać równoważnie w formie ograniczenia na błąd prawdziwy — z prawdopodobieństwem przynajmniej

$1-\delta$ dla każdej funkcji $f \in F$ mamy

$$\mathrm{er}_P(f) \leqslant \widehat{\mathrm{er}}_{\mathbf{z}}(f) + \sqrt{\frac{h\,(1+\ln(2m/h)) - \ln(\delta/4)}{m/8}}. \tag{6.29}$$

6.3.4 Funkcje rzeczywiste w uczeniu, pokrycia oraz liczby pokryciowe

W przypadku nieskończonych zbiorów funkcji zero-jedynkowych wykorzystywany był fakt, że dla każdej ustalonej próby zbiór funkcji odciętych do próby $(l_F)_{|\mathbf{z}_1,\ldots,\mathbf{z}_m}$ stawał się zbiorem skończonym. W uczeniu za pomocą funkcji rzeczywistych zbiór ten jest niestety nadal nieskończony — dziedzinę stanowi skończona liczba punktów, ale na przeciwdziedzinie nadal mamy continuum wartości. To uniemożliwia zliczanie. Potrzebnym zabiegiem staje się zastosowanie pojęcia ***pokrycia*** (ang. *cover*), co pozwala na redukcję zbioru nieskończonego do skończonego i w efekcie na zliczanie.

W ogólności mówimy, że zbiór U jest *ε-pokryciem* zbioru W zawartego w przestrzeni metrycznej, jeżeli dla każdego $w \in W$ istnieje element $u \in U$, taki że: $d(u,w) < \varepsilon$ (gdzie d oznacza przyjętą metrykę, czyli funkcję odległości). W problemach uczenia interesuje nas pokrywanie zbioru $(l_F)_{|\mathbf{z}_1,\ldots,\mathbf{z}_m}$, który stanowi pewne *zamazanie* zbioru $\mathbb{R}^m$. Jeżeli funkcje l_f są ograniczone, to mówimy o zamazaniu pewnej kostki zawartej w $\mathbb{R}^m$. Należy dodać, że istnieją twierdzenia, które pozwalają wyrazić pokrycie zbioru $(l_F)_{|\mathbf{z}_1,\ldots,\mathbf{z}_m}$ w terminach pokrycia zbioru $F_{|\mathbf{x}_1,\ldots,\mathbf{x}_m}$. Jest to udogodnienie, ponieważ zbiorem F zajmujemy się bezpośrednio.

Definicja 6.3.3 Liczbą pokryciową $\mathcal{N}(\varepsilon, F_{|\mathbf{x}_1,\ldots,\mathbf{x}_m}, d)$ nazywamy rozmiar minimalnego ε-pokrycia zbioru $F_{|\mathbf{x}_1,\ldots,\mathbf{x}_m}$ w metryce d.

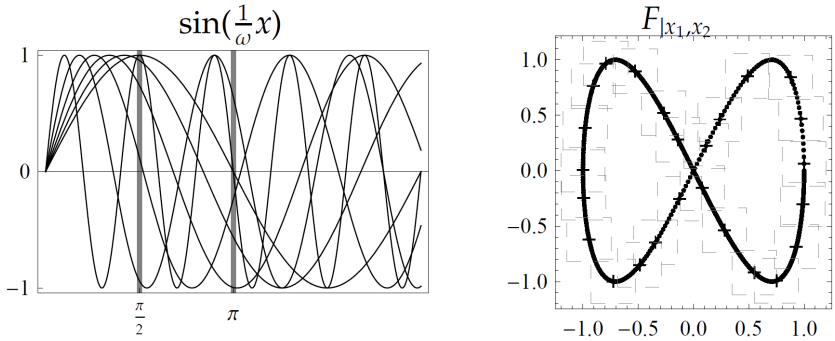
Definicja 6.3.4 Jednostajną liczbą pokryciową $\mathcal{N}_d(\varepsilon, F, m)$ nazywamy maksymalną spośród liczb $\mathcal{N}(\varepsilon, F_{|\mathbf{x}_1,\ldots,\mathbf{x}_m}, d)$ biorąc pod uwagę wszystkie możliwe próby o danym rozmiarze m:

$$\mathcal{N}_d(\varepsilon, F, m) = \max\{\mathcal{N}\left(\varepsilon, F_{|\mathbf{x}_1,\ldots,\mathbf{x}_m}, d\right) :\ \mathbf{x} \in \mathbf{X}^m\}. \tag{6.30}$$

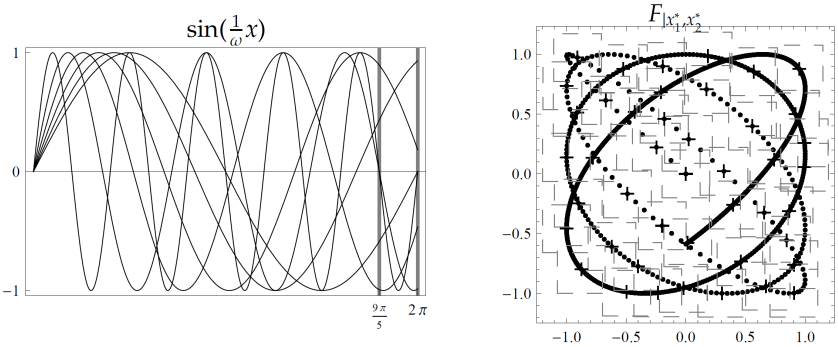
Należy zaznaczyć, że dla liczb pokryciowych używa się w ogólności metryk postaci

$$d_q(u,w) = \left(1/m \sum_i |u_i - v_i|^q\right)^{1/q}. \tag{6.31}$$

Z uwagi na czynnik $1/m$ zachodzi relacja: $\mathcal{N}_1(\cdot) \leqslant \mathcal{N}_2(\cdot) \leqslant \mathcal{N}_\infty(\cdot)$.



Rys. 6.8: Wykres kilku funkcji ze zbioru $F = \{\sin(\frac{1}{\omega}x) \colon \omega \in [0.2, 1]\}$ (lewa strona) oraz oraz przykładowe ε-pokrycie zbioru $F_{|\pi/2,\pi}$ — czyli pokrycie zamazania generowanego przez ten zbiór nad odciętymi $x_1 = \pi/2$, $x_2 = \pi$ (prawa strona). Pokrycie wyznaczone dla metryki d_∞ i $\varepsilon = 0.2$, zaznaczone na rysunku szarymi przerywanymi kwadratami (źródło: *opracowanie własne*).



$$\mathcal{N}_\infty(0.2, F, 2) \approx 44$$

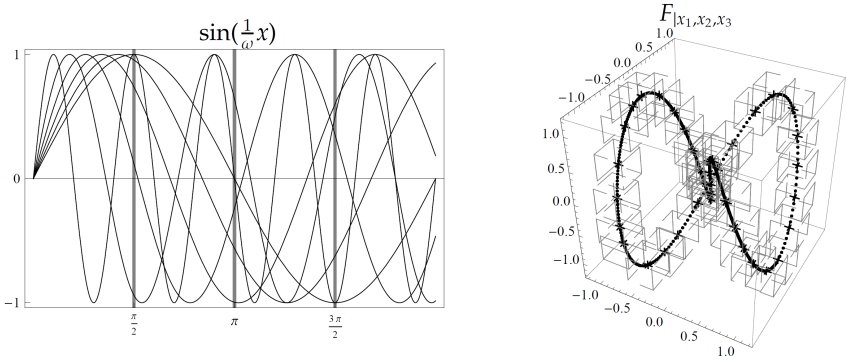
Rys. 6.9: Wykres kilku funkcji ze zbioru $F = \{\sin(\frac{1}{\omega}x) \colon \omega \in [0.2, 1]\}$ (lewa strona) oraz oraz przykładowe ε-pokrycie zbioru $F_{|9/5\pi,2\pi}$ — czyli pokrycie zamazania generowanego przez ten zbiór nad odciętymi $x_1^* = 9/5\pi$, $x_2^* = 2\pi$ (prawa strona). Pokrycie wyznaczone dla metryki d_∞ i $\varepsilon = 0.2$, zaznaczone na rysunku szarymi przerywanymi kwadratami. Szacowana jednostajna liczba pokryciowa wynosi 44 (źródło: *opracowanie własne*).

Aby dać Czytelnikowi wizualne wyobrażenie o pojęciach ε-pokrycia, liczby pokryciowej i jednostajnej liczby pokryciowej przedstawiamy rysunki 6.8–6.11, na których rozważany jest następujący zbiór funkcji trygonometrycznych

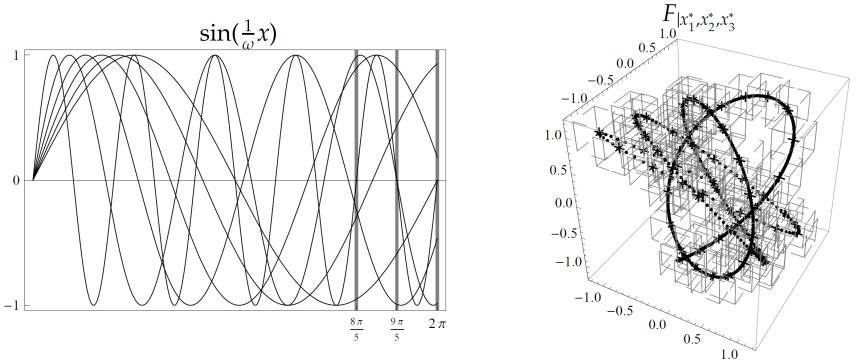
$$F = \left\{\sin\left(\frac{1}{\omega}x\right) \colon \omega \in [0.2, 1]\right\}, \tag{6.32}$$

z jednym parametrem ω decydującym o częstotliwości. Przyjęta jest metryka d_∞.

rys. 6.8 ilustruje przykładowe pokrycie zamazania generowanego przez zbiór F odciętego do dwuelementowej próby $x_1 = \pi/2,\ x_2 = \pi$. Rys. 6.9 prezentuje pokrycie zamazania wygenerowanego dla najbardziej wymagającej dwuelementowej próby $x_1^* = 9/5\pi,\ x_2^* = 2\pi$ (wykrytej poprzez przeszukiwanie wyczerpujące z krokiem $\pi/16$). Zgodnie z rysunkiem wykryta jednostajna liczba pokryciowa wynosi $N_\infty(0.2, F, 2) \approx 44$. Rysunki 6.10, 6.11 pokazują analogiczne przykłady dla prób trzyelelementowych.



Rys. 6.10: Wykres kilku funkcji ze zbioru $F = \{\sin(\frac{1}{\omega}x) : \omega \in [0.2, 1]\}$ (lewa strona) oraz oraz przykładowe ε-pokrycie zbioru $F_{|\pi/2,\pi,3/2\pi}$. Pokrycie wyznaczone dla metryki d_∞ i $\varepsilon = 0.2$, zaznaczone na rysunku szarymi przerywanymi sześcianami (źródło: *opracowanie własne*).



$$\mathcal{N}_\infty(0.2, F, 3) \approx 66$$

Rys. 6.11: Wykres kilku funkcji ze zbioru $F = \{\sin(\frac{1}{\omega}x) : \omega \in [0.2, 1]\}$ (lewa strona) oraz oraz przykładowe ε-pokrycie zbioru $F_{|8/5\pi, 9/5\pi, 2\pi}$. Pokrycie wyznaczone dla metryki d_∞ i $\varepsilon = 0.2$, zaznaczone na rysunku szarymi przerywanymi sześcianami. Szacowana jednostajna liczba pokryciowa wynosi 66 (źródło: *opracowanie własne*).

Istnieją następujące ważne twierdzenia o zbieżności jednostajnej wykorzystujące liczby pokryciowe.

Twierdzenie 6.3.2 (Anthony i Bartlett, [AB09, twierdzenie 10.1]) Niech F oznacza zbiór funkcji $f\colon \mathbf{X} \to [0,1]$, a P łączny rozkład prawdopodobieństwa zdefiniowany nad $\mathbf{X} \times [0,1]$. Niech: $0 < \varepsilon < 1$, $\gamma > 0$. Wtedy

$$P_m\left(\sup_{f\in F} \mathrm{er}_P(f) - \widehat{\mathrm{er}}^{\gamma}_{\mathbf{z}}(f) \geqslant \varepsilon\right) \leqslant 2\mathcal{N}_\infty(\gamma/2, F, 2m)e^{-\varepsilon^2 m/8}. \tag{6.33}$$

Twierdzenie 6.3.3 (Anthony i Bartlett, [AB09, twierdzenie 17.1]) Niech F oznacza zbiór funkcji $f\colon \mathbf{X} \to [0,1]$, a P łączny rozkład prawdopodobieństwa zdefiniowany nad $\mathbf{X} \times [0,1]$. Niech: $0 < \varepsilon < 1$. Wtedy

$$P_m\left(\sup_{f\in F} |\mathrm{er}_P(f) - \widehat{\mathrm{er}}_{\mathbf{z}}(f)| \geqslant \varepsilon\right) \leqslant 4\mathcal{N}_1(\varepsilon/16, F, 2m)e^{-\varepsilon^2 m/32}. \tag{6.34}$$

Twierdzenie 6.3.2 jest sformułowane dla zadania klasyfikacji postawionego jako *klasyfikacja z marginesem* γ i wykorzystuje liczbę pokryciową w metryce d_∞.

Margines reprezentuje odległość od progowej granicy decyzji, przy czym odległość ta jest liczona na osi wartości funkcji f, tzn.: $\text{margin}(f(\mathbf{x}),y) = f(\mathbf{x}) - \frac{1}{2}$ dla $y = 1$ oraz $\text{margin}(f(\mathbf{x}),y) = \frac{1}{2} - f(\mathbf{x})$ dla $y = 0$. Intuicyjnie: im większy margines, tym pewniejsze zaklasyfikowanie [Bar98]. W twierdzeniu pojawia się $\widehat{\text{er}}^{\gamma}_{\mathbf{z}}$, co oznacza częstość marginesu mniejszego niż γ na próbie, tj.: $\widehat{\text{er}}^{\gamma}_{\mathbf{z}}(f) = \frac{1}{m}\sum_{i=1}^{m}[\text{margin}\,(f(\mathbf{x}_i),y_i) < \gamma]$. Marginesu w powyższym rozumieniu nie należy mylić z pojęciem *marginesu separacji* w maszynach SVM. Tam margines liczony jest w przestrzeni $\mathbf{X}$ a nie na osi wartości funkcji f. Twierdzenie 6.3.3 jest sformułowane dla zadania estymacji regresji i wykorzystuje liczbę pokryciową w metryce d_1.

Następujące rezultaty są trzema przykładowymi ograniczeniami na liczby pokryciowe. Dwa pierwsze z nich wykorzystują *pseudowymiar*[15] (ang. *pseudodimension*) jako pojęcie pojemności zbioru funkcji. Ostatni rezultat (twierdzenie 6.3.6) wyraża liczbę pokryciową w terminach uczenia z ***regularyzacją*** dla funkcji liniowych ze względu na parametry. Regularyzacja powoduje, że oprócz błędu na próbie minimalizujemy także normę parametrów $\|w\|$ w pewnej metryce.

Twierdzenie 6.3.4 (Haussler i Long, [HL95]) Niech F oznacza zbiór funkcji rzeczywistych $f\colon\ \mathbf{X} \to [0,1]$ o pseudowymiarze równym h. Wtedy:

$$\mathcal{N}_{\infty}(\varepsilon,F,m) \leqslant \sum_{i=0}^{h}\binom{m}{i}\lfloor 1/\varepsilon\rfloor^{i}, \tag{6.35}$$

co z kolei jest mniejsze niż $\left(\frac{me}{\varepsilon h}\right)^{h}$ dla $m \geqslant h$.

Twierdzenie 6.3.5 (Haussler, [Hau95]) Niech F oznacza zbiór funkcji rzeczywistych $f\colon\ \mathbf{X} \to [0,1]$ o pseudowymiarze równym h. Wtedy:

$$\mathcal{N}_{1}(\varepsilon,F,m) \leqslant e(h+1)\left(\frac{2e}{\varepsilon}\right)^{h}. \tag{6.36}$$

Twierdzenie 6.3.6 (Zhang, [Zha02]) Niech F będzie zbiorem funkcji liniowych postaci: $f(\mathbf{x}) = \sum_{j=1}^{d} w_j x_j$, i niech algorytm uczący $\mathscr{L}_q$-regularyzuje wagi, tj. mamy, że $\|w\|_q \leqslant a$. Dla ustalonego q, zbiór danych jest znormalizowany

[15] Pseudowymiar jest tym dla funkcji rzeczywistych, czym wymiar VC dla funkcji zero-jedynkowych — w praktyce można utożsamiać te dwa pojęcia, a ich liczbowa wartość jest taka sama po zaokrągleniu rzeczywistych wartości f do $\{0,1\}$.

następująco: $\|\mathbf{x}_i\|_p \leqslant b$, $i = 1,\ldots,m$, gdzie $1/p + 1/q = 1$ (normy sprzężone) oraz $2 \leqslant p \leqslant \infty$. Wtedy:

$$\mathscr{N}_2(\varepsilon, F, m) \leqslant (2n+1)^{\lceil a^2 b^2/\varepsilon^2 \rceil}. \tag{6.37}$$

Twierdzenie 6.3.6 to bardzo atrakcyjny wynik. Po przełożeniu na złożoność próbkową mówi on, że przy zastosowaniu regularyzacji, do uczenia się z precyzją (ε, δ) wystarcza próba o rozmiarze proporcjonalnym tylko do logarytmu z liczby atrybutów (a nie skalująca się liniowo wraz z liczbą atrybutów). Przypomnijmy, że w wyrażeniu na złożoność próbkową pojawi się wyraz $\ln \mathscr{N}_1(\cdot)$ oraz że $\mathscr{N}_1(\cdot) \leqslant \mathscr{N}_2(\cdot)$. I tak po zlogarytmowaniu (6.37) otrzymamy $\lceil a^2 b^2/\varepsilon^2 \rceil \ln(2n+1) \sim O(\log n)$, zaś po zlogarytmowaniu (6.36) otrzymamy $(n+1)\ln(2e/\varepsilon) + \ln(n+2) + 1 \sim O(n)$, wiedząc, że wymiar VC wynosi $h = n+1$ dla funkcji liniowo zależnych od parametrów.

III Optymalizacja genetyczna

7. Metody gradientowe vs metody bez gradientu

Metody optymalizacji gradientowej, pomimo przeróżnych wariantów i usprawnień (m.in. omówionych w kontekście sieci neuronowych intensywnie w sekcjach 4.3.3—4.3.9), można z dużą dozą prawdziwości nazwać ogólnie metodami *przeszukiwania lokalnego*. Wynika to z faktu, że rozpoczynamy od pewnego punktu o wylosowanych współrzędnych (w wielowymiarowej przestrzeni parametrów), a następnie przeszukujemy, dokonując przejść: punkt stary przechodzi w punkt nowy, kierując się w dużej mierze informacją o gradiencie — czyli wektorze pochodnych cząstkowych. Takie postępowania powoduje, że trajektoria przeszukująca pozostaje na ogół w lokalnym zasięgu wpływów pewnego lokalnego optimum, w którym znalazł się punkt startowy. Oczywiście może zdarzyć się tak, że to optimum lokalne będzie jednocześnie globalnym, jednakże wcale nie musi [Klę05].

Wychodząc poza kontekst sieci neuronowych, dodatkowym mankamentem metod gradientowych jest sam fakt konieczności dysponowania informacją o pochodnych. Istnieje wiele problemów optymalizacyjnych, gdzie nie umiemy obliczać pochodnych (gradientu) lub wręcz takowe nie istnieją w sensie matematycznym. Często są to problemy o naturze dyskretnej, kombinatorycznej, ale nie tylko. Z tego punktu widzenia, atrakcyjniejsze na potrzeby optymalizacji wydają się być metody, które nie wymagają pochodnych, a jedynie możliwości obliczenia wartości optymalizowanej funkcji w dowolnym punkcie. Klasa takich metod, określona

angielską nazwą *derivative-free optimization*, obejmuje m.in.:

- metody spadku lub wspinaczki po współrzędnych (ang. *coordinate descent / ascent*) [Wri15],
- metodę poszukiwania wzorców (ang. *pattern search*) [Dav91; HJ61],
- symulowane wyżarzanie (ang. *simulated annealing*) [Met+53],
- optymalizację za pomocą roju cząstek (ang. *particle swarm optimization* — PSO) [KE95],
- algorytmy ewolucyjne (ang. *evolutionary algorithms* — EA), w tym w szczególności **algorytmy genetyczne** (ang. *genetic algorithms* — GA) [Gol89; Hol75].

To właśnie tym ostatnim poświęcony jest ten rozdział niniejszego podręcznika.

Wiele spośród wymienionych powyżej metod posiada znamiona *przeszukiwania globalnego* i istnieje wiele przykładów problemów, czy też benchmarków, gdzie aplikowanie tych metod pozwoliło na wykrycie optimum globalnego. Pomimo tych skutecznych przykładów „na rzecz", należy uczciwie zaznaczyć, że istnieją również kontrprzykłady. A co z tego wynika — żadna z powyższych metod nie gwarantuje w sensie matematycznym umiejętności wykrycia optimum globalnego dla *dowolnego* problemu optymalizacyjnego.

Biorąc pod uwagę dwa ostatnie spośród wymienionych podejść — PSO oraz EA/GA, niewątpliwą ich zaletą jest przeszukiwanie, które można nazwać *populacyjnym* (czy też gromadnym lub zespołowym). Oznacza te, że przestrzeń parametrów nie jest przeszukiwana za pomocą pojedynczej trajektorii (punkt $\rightarrow$ punkt $\rightarrow \ldots$), a za pomocą zbioru punktów (populacji kandydackich rozwiązań), który w kolejnych iteracjach przechodzi w nowy zbiór punktów itd. Jest to atrakcyjne, ponieważ przestrzeń rozwiązań jest próbkowana w wielu miejscach równocześnie i zwykle nie ma potrzeby wielokrotnego startu algorytmu, jak ma to miejsce w metodach gradientowych, a co więcej stwarza to naturalne możliwości obliczeniowe do zrównoleglania takich algorytmów (m.in. z wykorzystaniem procesorów graficznych ogólnego przeznaczenia i technologii CUDA). Ponadto zaletą wymienionych algorytmów jest minimalna znajomość rozwiązywanego problemu, gdyż aby wykorzystać algorytmy populacyjne, trzeba zdefiniować kodowanie kandydatów na rozwiązanie oraz funkcję celu optymalizacji.

7.1 Algorytm genetyczny

Algorytm genetyczny jest to rodzaj heurystyki przeszukującej przestrzeń alternatywnych rozwiązań problemu w celu wyszukania rozwiązań najlepszych [Gwi07; Gwi09; Rut12]. Metoda została zaproponowana przez Johna H. Hollanda w roku 1975 [Hol75]. Duży wkład w jej rozwój miał David E. Goldberg, który w 1989 opublikował książkę pt. *„Genetic Algorithms in Search, Optimization and Machine*

Learning" [Gol89]. Sposób działania algorytmów genetycznych przypomina zjawisko ewolucji biologicznej i jest zaliczany do grupy algorytmów ewolucyjnych. Algorytmy ewolucyjne są procedurami opartymi na zasadach działania doboru naturalnego i dziedziczenia. Do tej grupy algorytmów możemy oprócz algorytmu genetycznego zaliczyć metody takie jak: strategie ewolucyjne, programowanie ewolucyjne czy programowanie genetyczne. Algorytmy należące do tej grupy wyróżniają się od tradycyjnych metod optymalizacji następującymi cechami:

- parametry zadania są przetwarzane w postaci zakodowanej,
- działają na populacji możliwych rozwiązań,
- korzystają z możliwie małej ilości informacji na temat zadania zapisanej w postaci funkcji celu,
- stosują probabilistyczne metody wyboru,
- stosują probabilistyczne metody modyfikacji osobników lub wymiany cząstkowych informacji pomiędzy nimi.

Algorytmy genetyczne (tak jak wszystkie algorytmy ewolucyjne) korzystają z nazw pojęć zapożyczonych z genetyki. Poniżej przedstawiamy zestaw podstawowych takich pojęć.

- **Populacja** to zbiór osobników o określonej liczebności.
- **Osobnik** to zakodowane rozwiązanie w postaci chromosomu(ów). Każdy z osobników reprezentuje pewne (lepsze lub gorsze) rozwiązanie danego problemu. Każdy z osobników może posiadać kilka chromosomów, jednak bardzo często w algorytmach genetycznych wybrany osobnik posiada jeden chromosom. W literaturze oba pojęcia: chromosom oraz osobnik bardzo często używane są zamiennie.
- **Chromosom** to ciąg genów (lub dowolnie uporządkowany zbiór genów).
- **Gen** to podstawowa jednostka informacji w chromosomie. Stanowi on pojedynczy element genotypu.
- **Genotyp** to zespół chromosomów danego osobnika. Stanowi on podstawę do utworzenia fenotypu.
- **Fenotyp** to zestaw objawiających się cech / własności wysokopoziomowych danego osobnika zakodowanych niskopoziomowo popprzez dany genotyp. Fenotyp podlega ocenie poprzez funkcję przystosowania.
- **Allel** to wartość danego genu.
- **Locus**[1] to pozycja (indeks) wskazująca położenie danego genu w chromosomie.
- **Funkcja przystosowania** (ang. *fitness function*) to funkcja służąca do oceny ilościowej mówiącej, jak dobrze dany osobnik jest przystosowany. Funkcja przystosowania jest zdefiniowana przez problem, który należy rozwiązać.

[1] Liczba mnoga to *loci*.

Innymi słowy, genotyp opisuje (reprezentuje, koduje) proponowane rozwiązanie problemu, a funkcja przystosowania ocenia, na ile dobre jest to rozwiązanie. Funkcja przystosowania pozwala na wybranie najlepszych osobników, zgodnie z ewolucyjną zasadą przetrwania „najsilniejszych". W zagadnieniach optymalizacyjnych funkcja przystosowania jest z reguły funkcją celu. Klasyczny algorytm genetyczny stosuje się do zadania maksymalizacji. W przypadku zadań minimalizacji funkcji celu bardzo często przekształca się je do problemu maksymalizacji (np. poprzez dostawienie znaku minus).

– **Generacja (pokolenie)** to populacja osobników w danej iteracji algorytmu.

Pseudokod zapisany jako Algorytm 11 przedstawia wysokopoziomowe kroki klasycznego algorytmu genetycznego (szczegóły tych kroków omówione zostaną później). Poza jawnym argumentem T oznaczającym liczbę iteracji, dla uproszczenia zapisu podano tylko hasłowo „pozostałe nastawy". Rzecz w tym, że zwyczajowo w aglorytmie może to być długa lista nastaw zawierająca m.in.: rozmiar populacji, wskaźnik na funkcję przystosowania (jednocześnie definijującą sam problem), prawdopodobieństwo krzyżowania, prawdopodobieństwo mutacji, nazwy metod (lub wskaźniki na odpowiednie funkcji) selekcji, krzyżowania, mutacji, flaga logiczna elitaryzmu, dodatkowe warunki stopu.

Algorytm 11 Algorytm genetyczny

```
1: procedura AlgorytmGenetyczny(T, pozostałe nastawy)        ▷ T — liczba iteracji
2:     wygenerowanie początkowej populacji osobników
3:     dla t = 1,...,T wykonaj
4:         ocena osobników za pomocą funkcji przystosowania
5:         selekcja
6:         krzyżowanie
7:         mutacja
8:     zwróć „najlepszego" osobnika          ▷ z ostatniego pokolenia lub z całej historii
```

Należy pamiętać, że algorytm genetyczny nie przetwarza pojedynczego rozwiązania, ale cały zbiór nazywany populacją. W wyniku działania algorytmu otrzymujemy najlepsze wykryte rozwiązanie (czasami jest to zbiór rozwiązań). Oczywiście zwrócone rozwiązanie nie musi być tożsame z rozwiązaniem optymalnym matematycznie, a bywa tylko jego przybliżeniem. O tym, jak często algorytm genetyczny będzie w stanie wykryć optymalne rozwiązanie, decyduje wiele elementów: łatwość / trudność samego problemu, rozmiar populacji, stan populacji początkowej, jakość zaprojektowanej funkcji przystosowania, liczba iteracji algorytmu, wybrane operacje genetyczne, inne nastawy ilościowe (w tym probabilistyczne).

7.1.1 Generowanie populacji początkowej

Generowanie populacji początkowej polega na losowym lub deterministycznym wyborze zadanej liczby osobników o określonej długości chromosomu(ów). W klasycznej wersji algorytmu rozmiar populacji jest stały. Niezmienna jest również długość chromosomów, która jest ściśle związana z problemem, a dokładniej z potrzebną długością reprezentacji rozwiązania na rzecz tego problemu. Współcześnie stosowane są także inne warianty kształtu populacji i dopuszczają one między innymi: zmienną liczbę osobników w populacji, podział populacji na grupy. Pojedynczy chromosom może zawierać informacje zakodowane:

- liczbami binarnymi, np.: $(1,1,0,1,0,0,1)$,
- liczbami zmiennoprzecinkowymi, np.: $(1.23, 1.11, 4.26, 9, 9.04)$,
- liczbami całkowitymi, np.: $(2,3,1,4,5,14)$,
- symbolami z dowolnego alfabetu (w praktyce sprowadza się do kodowania liczbami całkowitymi).

W niniejszym skrypcie przybliżymy jedynie problemy oraz operatory genetyczne używające kodowania liczbami binarnymi i całkowitymi, przy zastrzeżeniu, że przy kodowaniu liczbami całkowitymi wartości poszczególnych genów nie mogą się powtarzać w danym chromosomie (stanowią permutację n liczb).

7.1.2 Sprawdzenie warunków zatrzymania

Podstawowym warunkiem zatrzymania się klasycznego algorytmu genetycznego jest osiągnięcie zadanej liczby kroków (iteracji). Jednocześnie lub alternatywnie można stosować inne warunki stopu, takie jak:

- upłynięcie określonego czasu od momentu rozpoczęcia działania algorytmu,
- brak poprawy rozwiązania w czasie działania algorytmu przez dany okres (liczbę iteracji),
- osiągnięcie zadawalającej zadanej z góry wartości przystosowania.

Ostatni warunek ma zastosowanie w zadaniach optymalizacji, gdy znana jest maksymalna (lub minimalna) wartość funkcji przystosowania.

7.1.3 Ocena przystosowania osobników w populacji

Funkcja przystosowania ocenia, na ile dobrze dany osobnik jest przystosowany z punktu widzenia rozwiązywanego problemu. Musi ona zostać obliczona dla każdego kandydackiego rozwiązania czyli dla każdego osobnika w populacji (i dzieje się to w każdej iteracji algorytmu). Zwykle dobrze jest, gdy funkcja ta w sposób dokładny odzwierciedla treść zadania optymalizacyjnego, które chcemy rozwiązać. W przypadkach gdy jest to z jakiegoś powodu utrudnione lub np. gdy próbujemy optymalizować klika wielkości jednocześnie (czasami stojących w sprzeczności), dopuszcza się, aby projektant algorytmu genetycznego zaproponował

własną heurystyczną postać funkcji przystosowania, co jest obarczone różnymi ryzykami. Przykładowe funkcje przystosowania na rzecz dyskretnego problemu plecakowego oraz problemu komiwojażera zostaną omówione w punktach 7.2.1 i 7.2.2.

7.1.4 Selekcja osobników

Etap selekcji ma za zadanie wybranie najlepiej przystosowanych osobników i umieszczenie ich w nowym pokoleniu w większej liczbie egzemplarzy w stosunku do poprzedniego pokolenia. Oznacza to równocześnie, że osobniki najsłabsze powinny zniknąć z populacji (robiąc miejsce dla tych powielonych najlepszych). Wybór osobników powinien zawsze odbywać się zgodnie z zasadą selekcji naturalnej, czyli największe szanse na sukcesję — przejście do następnego pokolenia — powinny mieć osobniki o największych wartościach funkcji przystosowania.

Uważa się , że dobra metoda selekcji powinna także mieć własność zachowywania możliwie dużej różnorodności genetycznej w populacji. Ta własność pozwala zapobiegać zbyt szybkiej zbieżności algorytmu do pewnego optimum lokalnego (co może zdarzyć się poprzez zbieżność populacji do chwilowo najlepszego osobnika).

Selekcja koła ruletki

Selekcja koła ruletki polega na budowaniu wirtualnego koła do przeprowadzania losowań, którego wycinki odpowiadają poszczególnym osobnikom. W tej metodzie rozmiary tych wycinków są *wprost propocjonalne* do wartości funkcji oceny. Tym samym, im lepszy osobnik, tym większy wycinek koła zajmuje i ma większe prawdopodobieństwo na bycie wylosowanym. Przy populacji złożonej z m osobników, prawdopodobieństwo dla osobnika x_i (czyli zajętość koła przez jego fragment) definiujemy zgodnie ze wzorem:

$$P(x_i) = \frac{f(x_i)}{\sum_{j=1}^{m} f(x_j)}. \tag{7.1}$$

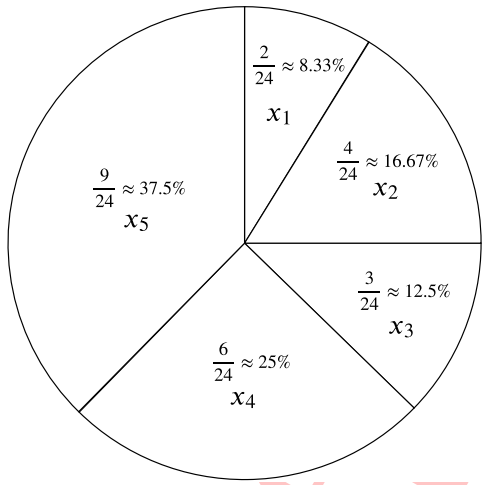
Dla przykładu, mając 5 chromosomów o następujących wartościach funkcji przystosowania:

$$f(x_1) = 2, \quad f(x_2) = 4, \quad f(x_3) = 3, \quad f(x_4) = 6, \quad f(x_5) = 9,$$

otrzymujemy następujące prawdopodobieństwo wylosowania poszczególnych osobników:

$$P(x_1) = \frac{2}{24}, \quad P(x_2) = \frac{4}{24}, \quad P(x_3) = \frac{3}{24}, \quad P(x_4) = \frac{6}{24}, \quad P(x_5) = \frac{9}{24}.$$

Koło ruletki zbudowane dla powyższego przykładu zostało pokazane na Rys. 7.1.



Rys. 7.1: Koło ruletki — przykładowy podział (źródło: *opracowanie własne*).

Przeprowadzenie całego etapu selekcji polega zatem na wykonaniu m losowań i każdorazowym przenoszeniu do nowego pokolenia egzemplarza osobnika, w którego fragment trafił wynik pojedynczego losowania. W implementacjach zwyczajowo losowania odbywają się nad przedziałem $(0,1)$ zgodnie z działaniem typowych funkcji randomizujących, co można równoważnie rozumieć jako „rozprostowany" obwód wspomnianego wirtualnego koła ruletki. Punkty graniczne wyznaczające podprzedziały kolejnych osobników w ramach przedziału $(0,1)$ są określone przez ciąg sum: 0, $P(x_1)$, $P(x_1)+P(x_2)$, $P(x_1)+P(x_2)+P(x_3)$, …, $P(x_1)+\cdots+P(x_m)=1$.

Selekcja rankingowa

Selekcję rankingową (zwaną także rangową) można rozumieć jako odmianę selekcji ruletkowej, w której prawdopodobieństwa sukcesji nie są wprost proporcjonalne do wartości funkcji przystosowania, zaś do *pozycji w rankingu* przystosowań — czyli pozycji w ciągu przystosowań posortowanych od najmniejszego do największego. Rangi zwykle ustala się jako kolejne liczby naturalne, czyli $\{1,2,\ldots,m\}$, gdzie m reprezentuje rangę najlepszego osobnika (o największej wartości funkcji przystosowania). Tym samym każdemu osobnikowi przydzielane jest prawdopodobieństwo wylosowania zgodne ze wzorem:

$$P(x_i) = \frac{\operatorname{rank}(f(x_i))}{\sum_{j=1}^{m} \operatorname{rank}(f(x_j))} = \frac{2\operatorname{rank}(f(x_i))}{m(m+1)}. \tag{7.2}$$

W drugiej postaci wykorzystano fakt, że suma rang w mianowniku jest sumą ciągu arytmetycznego $1+2+\cdots+m$ wynoszącą $m(m+1)/2$.

Selekcja turniejowa (turniej o rozmiarze k)

Selekcja turniejowa również zawiera elementy losowe, ale nie przeprowadza losowań na kole ruletki, zaś stara się luźno naśladować mechanizmy znane ze sportowych rozgrywek turniejowych. Dla turnieju o zadanym rozmiarze k, polega ona na wylosowaniu *bez powtórzeń* grupy k-elementowej z populacji, a następnie wyborze zwycięzcy turnieju czyli osobnika o największym przystosowaniu. Taką operację — zaaranżowanie pojedynczego turnieju — powtarzamy m razy (zgodnie z zadanym rozmiarem populacji).

Zwykle przeprowadza się małe turnieje, najczęściej wybierając $k=2$. Warto uzmysłowić sobie, że przeciwna skrajność $k=m$ skutkuje (w związku z losowaniami bez powtórzeń) wypełnieniem populacji wynikowej m egzemplarzami najlepszego aktualnie osobnika. A taka sytuacja skutecznie niszczy różnorodność populacji i prowadzi do przedwczesnej zbieżności algorytmu.

Ciekawym pobocznym zagadnieniem matematycznym w selekcji turniejowej jest ryzyko utraty najlepszego osobnika obecnego w populacji. Taka sytuacja może mieć miejsce, gdy najlepszy osobnik ze względu na losowania uczestników turniejów nie dostał się do żadnego z nich. Należy zwrócić uwagę, że losowania te są niezależne od wartości funkcji przystosowania. Pradopodobieństwo (ryzko) takiej sytuacji dla $k=2$ wynosi:

$$\left(\frac{m-2}{m}\right)^m = \left(1-\frac{2}{m}\right)^m \approx \frac{1}{e^2} \approx 0.1353. \tag{7.3}$$

Przy przejściu do liczby $1/e^2$ zakładamy odpowiednio duży rozmiar populacji. Można sprawdzić, że już dla $m=100$ otrzymujemy ≈ 0.1326. A zatem jest to ryzyko stosunkowo duże. Aby uchronić się przed taką sytuacją, w algorytmach genetycznych z selekcją turniejową (ale nie tylko taką) stosuje się czasem prosty zabieg zastąpienia osobnika najsłabszego po selekcji osobnikiem najlepszym sprzed selekcji (jest to wariant ogólniejszego podejścia zwanego *elitaryzmem*).

7.1.5 Operatory genetyczne (krzyżowanie i mutacja)

Operatory genetyczne mają na celu rekombinację genów w chromosomach. Dwa najczęstsze operatory genetyczne to krzyżowanie i mutacja. Każdy z operatorów genetycznych wykonywany jest z pewnym prawdopodobieństwem, które definiujemy na początku programu, i które zazwyczaj[2] jest stałe w trakcie jego trwania. Typowe zakresy prawdopodobieństw dla operatorów genetycznych są następujące:

[2]Istnieją warianty algorytmów genetycznych oparte tylko na mutacji, w których rozpoczynamy od dużych prawdopodobieństw i stopniowo wygaszamy je do bliskich zeru w trakcie pracy algo-

– 0.5 – 1.0 dla krzyżowania,
– 0.0 – 0.1 dla mutacji (zwykle prawdopodobieństwo określone na poziomie pojedynczego genu).

W klasycznym algorytmie genetycznym kolejność zastosowania obu operatorów nie ma znaczenia — mutacji można dokonać na pokoleniu rodziców przed krzyżowaniem lub na pokoleniu potomków po krzyżowaniu.

Operator krzyżowania

Krzyżowanie (ang. *crossing-over* lub *crossover*), podobnie jak w naturze, jest operacją mającą na celu wymianę materiału genetycznego pomiędzy osobnikami. W procesie krzyżowania kojarzymy całą populację w losowe pary rodzicielskie. Następnie dla każdej z par dokonujemy krzyżowania z zadeklarowanym wcześniej prawdopodobieństwem, generując tym samym z pary rodziców parę potomków. W przypadku gdy nie jest stosowany operator krzyżowania, wartości genów rodziców są bezpośrednio kopiowane do potomków. Poniżej omawiamy kilka popularnych operatorów krzyżowania.

Krzyżowanie jednopunktowe (ang. *1-point crossover*) (1-PX)

Krzyżowanie to polega na wylosowaniu jednego punktu krzyżowania wewnątrz chromosomów, a następnie wymianie materiału genetycznego, która dokonuje się zgodnie z poniższym schematem (punkt krzyżowania oznaczono pionową kreską):

rodzice:		potomkowie:
$x_1 = (\mathbf{00110}\|\mathbf{011})$	$\xrightarrow{\text{krzyżowanie}}$	$(\mathbf{00110}\|101)$
$x_2 = (01101\|101)$		$(01101\|\mathbf{011})$.

Krzyżowanie wielopunktowe (ang. *multi-point crossover*) (k-PX)

Krzyżowanie wielopunktowe jest wykonywane analogicznie do krzyżowania jednopunktowego, z tą różnicą że losowanych jest więcej punktów krzyżowania. Poniżej znajduje się przykład krzyżowania dwupunktowego (2-PX):

rodzice:		potomkowie:
$x_1 = (\mathbf{001}\|\mathbf{10}\|\mathbf{011})$	$\xrightarrow{\text{krzyżowanie}}$	$(011\|\mathbf{10}\|101)$
$x_2 = (011\|01\|101)$		$(\mathbf{001}\|01\|\mathbf{011})$.

rytmu. W optymalizacji problemów ze zmiennymi ciągłymi, istnieją także warianty, w których prawdopodobieństwo mutacji zawsze wynosi 1, natomiast wygaszaniu ulega tzw. promień mutacji.

W przypadku rozwiązań kodowanych liczbami całkowitymi, w których wartości poszczególnych genów nie mogą się powtarzać w danym chromosomie (np. stanowią permutację n liczb), nie można zastosować powyższych operatorów. Doprowadziłby to do sytuacji, w której w danym chromosomie dwa różne geny miałyby tę samą wartość. Kodowanie tego typu ma miejsce np. w problemie komiwojażera, gdzie wartość każdego genu stanowi numer miasta. Przy powyższym typie chromosomu należy stosować operatory takie jak krzyżowanie: z zachowaniem porządku, z częściowym odwzorowaniem, cykliczne.

Krzyżowanie z częściowym odwzorowaniem (ang. *partially-mapped crossover*) (PMX)

Krzyżowanie z częściowym odwzorowaniem jest trochę bardziej skomplikowane. Proces rozpoczyna się od wylosowania dwóch punktów krzyżowania i przekopiowania części genów [GL85]:

$$\begin{matrix} x_1 = (\mathbf{58}|\mathbf{213}|\mathbf{764}) \\ x_2 = (78|462|531) \end{matrix} \xrightarrow{\text{krzyżowanie}} (- - |\mathbf{213}| - - -).$$

W analogicznym segmencie w x_2 znajdują się elementy, które nie zostały skopiowane. Należy je umieścić w potomku. Dla wylosowanych (tak jak powyżej) dwóch punktów krzyżowania mamy następujące odwzorowania: $4 \to 2$, $6 \to 1$, $2 \to 3$. Dla przykładu $6 \to 1$ oznacza, że należy skopiować wartość 6 w miejsce genu o wartości 1:

$$\begin{matrix} x_1 = (\mathbf{58}|\mathbf{213}|\mathbf{764}) \\ x_2 = (78|462|531) \end{matrix} \xrightarrow[6\to1]{\text{krzyżowanie}} (- - |\mathbf{213}| - - 6).$$

Podobnej operacji nie możemy wykonać dla $4 \to 2$, ponieważ 2 zostało już skopiowane. Dlatego dalej sprawdzamy, który gen został skopiowany na miejsce 4. Dokonujemy następującego przejścia $4 \to 2 \to 3$, czyli kopiujemy 4 w miejsce wartości 3:

$$\begin{matrix} x_1 = (\mathbf{58}|\mathbf{213}|\mathbf{764}) \\ x_2 = (78|462|531) \end{matrix} \xrightarrow[4\to2\to3]{\text{krzyżowanie}} (- - |\mathbf{213}| - 46).$$

Pozostałe geny kopiujemy z x_2:

$$\begin{matrix} x_1 = (\mathbf{58}|\mathbf{213}|\mathbf{764}) \\ x_2 = (78|462|531) \end{matrix} \xrightarrow{\text{krzyżowanie}} (78|\mathbf{213}|546).$$

Drugiego potomka generujemy zgodnie z powyższym opisem, zamieniając x_1 i x_2 miejscami:

rodzice:		potomkowie:
$x_1 = (\mathbf{58}\|\mathbf{213}\|\mathbf{764})$	$\xrightarrow{\text{krzyżowanie}}$	$(78\|\mathbf{213}\|546)$
$x_2 = (78\|462\|531)$		$(\mathbf{58}\|462\|\mathbf{713})$.

Krzyżowanie z zachowaniem porządku (ang. *order crossover*) (OX)

Podobnie jak w przypadku krzyżowania PMX, w krzyżowaniu OX losowane są dwa punktu w chromosomach rodziców [Dav85]. Do potomków przepisywane są fragmenty pomiędzy wylosowanymi punktami. Puste fragmenty uzupełniane są począwszy od drugiego punktu krzyżowania elementami z drugiego rodzica, które jeszcze nie są obecne w potomku.

Rozpisując przykład bardziej szczegółowo, na początku mamy kopiowanie środkowych fragmentów:

$x_1 = (\mathbf{58}\|\mathbf{213}\|\mathbf{764})$	$\xrightarrow{\text{krzyżowanie}}$	$(\ -\ -\ \|\mathbf{213}\|\ -\ -\ -\ )$
$x_2 = (78\|462\|531)$		$(\ -\ -\ \|462\|\ -\ -\ -\ )$.

Następnie geny z chromosomu x_2 układamy rozpoczynając od drugiego wylosowanego punktu (53178462), a następnie wykreślamy z niego elementy, które już znajdują się w pierwszym potomku (**213**), i otrzymujemy (57846). Podobnie postępujemy z genami z chromosomu x_1 (**76458213**), z których pomijamy geny znajdujące się w drugim potomku (462) co daje nam (**75813**). Tak wygenerowane ciągi genów wpisujemy do odpowiednich potomków rozpoczynając od drugiego punktu krzyżowania, w efekcie uzyskując:

rodzice:		potomkowie:
$x_1 = (\mathbf{58}\|\mathbf{213}\|\mathbf{764})$	$\xrightarrow{\text{krzyżowanie}}$	$(46\|\mathbf{213}\|578)$
$x_2 = (78\|462\|531)$		$(\mathbf{13}\|462\|\mathbf{758})$.

Krzyżowanie cykliczne (ang. *cycle crossover*) (CX)

Ten rodzaj krzyżowania kreuje potomków w ten sposób, że każdy gen wraz z jego miejscem pochodzi od jednego z rodziców [OSH87]. Na początek losujemy gen i rodzica, od którego zaczynamy krzyżowanie. Powiedzmy, że zaczniemy od wartości

5 w pierwszym rodzicu x_1:

$$\begin{array}{l} x_1 = (\mathbf{58213764}) \\ x_2 = (78462315) \end{array} \xrightarrow{\text{krzyżowanie}} (\mathbf{5}\text{ - - - - - - - }).$$

5 leży na tym samym miejscu co 7, więc do chromosomu dopisujemy **7** w tym miejscu, w którym było ono zapisane w pierwszym chromosomie:

$$\begin{array}{l} x_1 = (\mathbf{58213764}) \\ x_2 = (78462315) \end{array} \xrightarrow{\text{krzyżowanie}} (\mathbf{5}\text{ - - - - }\mathbf{7}\text{ - - }).$$

7 leży na tym samym miejscu co 3, dlatego do potomka dodajemy **3** w miejscu z pierwszego chromosomu:

$$\begin{array}{l} x_1 = (\mathbf{58213764}) \\ x_2 = (78462315) \end{array} \xrightarrow{\text{krzyżowanie}} (\mathbf{5}\text{ - - }\mathbf{37}\text{ - - }).$$

3 leży na tym samym miejscu co 2, zatem do chromosomu dodajemy **2**. Całość kontynuujemy aż do momentu zamknięcia się cyklu, czyli do natrafienia na gen, który już dodaliśmy do chromosomu (w poniższym przypadku to 5):

$$\begin{array}{l} x_1 = (\mathbf{58213764}) \\ x_2 = (78462315) \end{array} \xrightarrow{\text{krzyżowanie}} (\mathbf{5}\text{ - }\mathbf{2}\text{ - }\mathbf{37}\text{ - }\mathbf{4}).$$

Resztę genów uzupełniamy genami z chromosomu x_2:

$$\begin{array}{l} x_1 = (\mathbf{58213764}) \\ x_2 = (78462315) \end{array} \xrightarrow{\text{krzyżowanie}} (\mathbf{58263714}).$$

Dla drugiego potomka uzupełnianie zaczynamy od 7. A wolne miejsca pozostałe po zakończeniu cyklu uzupełniamy z chromosomu x_1:

Pokolenie rodziców:		Pokolenie potomków:
$x_1 = (\mathbf{58213764})$	$\xrightarrow{\text{krzyżowanie}}$	$(\mathbf{58263714})$
$x_2 = (78462315)$		$(\mathbf{78412365}).$

Operator mutacji

Mutacja ma za zadanie wprowadzić różnorodność genetyczną populacji. Dla chromosomów kodujących informację binarnie mutacja polega na zamianie wartości bitu na przeciwny, na przykład:

$$(00110010101) \xrightarrow{\text{mutacja}} (00110011101).$$

Prawdopodobieństwo dokonania mutacji można zastosować w dwóch wariantach. W pierwszym wariancie prawdopodobieństwo mutacji jest losowane dla całego chromosomu. W sytuacji gdy mutacja ma mieć miejsce, losuje się miejsce mutacji. W drugim wariancie prawdopodobieństwo mutacji jest losowane osobno dla każdego genu w każdym chromosomie.

Podobnie jak w przypadku krzyżowania, tak samo w przypadku mutacji dla problemów kodowanych liczbami całkowitymi nie można zastosować powyższego operatora. Alternatywnym podejściem jest zastosowanie *mutacji poprzez inwersję*. Inwersja polega na wylosowaniu podzakresu w chromosomie i wstawieniu genów w tym podzakresie w kolejności przeciwnej:

$$(12\mathbf{3456}789) \xrightarrow{\text{mutacja}} (12\mathbf{6543}789)$$

Alternatywnie można wybrać dany zakres genów w chromosomie (lub wybrać nawet pojedyncze geny) i dokonać losowej permutacji znajdujących się w nich wartości:

$$(12\mathbf{3456}789) \xrightarrow{\text{mutacja}} (12\mathbf{5364}789)$$

7.1.6 Utworzenie nowej populacji

Osobniki otrzymane w wyniku działania operatorów genetycznych stanowią nową populację. W następnej iteracji nowa populacja staje się populacją bieżącą i to właśnie na niej będzie pracował algorytm genetyczny opisany w poprzednich akapitach. W klasycznym algorytmie genetycznym nowa populacja zawsze jest tak samo liczna jak poprzednia.

7.1.7 Wyprowadzenie „najlepszego" chromosomu

Po zatrzymaniu algorytmu należy zwrócić wynik jego działania. Wynikiem jest najlepiej przystosowany chromosom, czyli taki, w którym wartości funkcji przystosowania jest największa. Istnieją tutaj dwa podejścia wyboru najlepszego chromosomu: najlepszy z ostatniego pokolenia oraz najlepszy w całej historii. Te dwa pojęcia nie muszą być sobie równoważne, ponieważ najlepsze rozwiązanie w całej historii, może zostać utracone w wyniku działania algorytmu (w wyniku selekcji lub któregoś z operatorów genetycznych). Wspomniany wcześniej „elitaryzmu" jest typowym zabiegiem przeciwdziałającym takiej sytuacji — w każdej iteracji zastępuje się najsłabszego osobnika poprzez dotychczasnajlepszego, aby go nie utracić.

7.2 Przykładowe problemy

Dla lepszego zrozumienia tematu poniżej zostaną przybliżone dwa klasyczne przykłady, w których postawione problemy rozwiązywane są przy pomocy algorytmu genetycznego: dyskretny problem plecakowy i problem komiwojażera.

7.2.1 Dyskretny problem plecakowy

Dyskretny problem plecakowy (ang. DKP — *Discrete Knapsack Problem*) często przedstawia się jako problem złodzieja rabującego sklep. Złodziej natrafia na n przedmiotów (towarów); i-ty przedmiot jest wart v_i oraz posiada określoną objętość c_i. Złodziej dąży do zabrania ze sobą jak najbardziej wartościowszego łupu, przy czym nie może przekroczyć objętości swojego plecaka wynoszącej C. Przedmiotów nie można dzielić, np. nie można wziąć kawałka telewizora. Możliwość dzielenia ma miejsce w *ciągłym* problemie plecakowym (inny wariant problemu) i przekłada się na intuicyjnie łatwe rozwiązanie zachłanne, w którym przedmioty są wybierane zgodnie z malejącą kolejnością proporcji v_i/c_i, a ostatni nie mieszczący się w całości przedmiot jest kawałkowany tak, aby dopełnić plecak. Można udowodnić, że takie podejście zachłanne prowadzi do rozwiązania optymalnego, ale tylko w problemie ciągłym. Nie działa ono w ogólności dla dyskretnego wariantu problemu, tzn. nie musi prowadzić do rozwiązania optymalnego. Wskazanie dowolnego kontrprzykładu pozostawiamy jako ćwiczenie dla Czytelnika.

Formalna definicja dyskretnego problemu plecakowego jest następująca. Do dyspozycji mamy n przedmiotów, a każdy z nich jest opisany jako para (v_i, c_i):

$$A = \{(v_i, c_i)\}_{i=1\ldots n}. \tag{7.4}$$

Zadaniem jest znalezienie takiego podzbioru B, w którym suma wartości przedmiotów jest maksymalna, a suma objętości nie przekracza objętości plecaka, czyli:

$$B \subseteq A, \qquad \sum_{(v_i,c_i)\in B} v_i \longrightarrow \max, \qquad \sum_{(v_i,c_i)\in B} c_i \leqslant C. \tag{7.5}$$

Na potrzeby niniejszego zadania użyjemy chromosomów binarnych, w których każdy z genów może przyjmować tylko jedną wartość 0 lub 1.

Każdy osobnik będzie reprezentował jedno możliwe rozwiązanie, tj. pewien podzbiór przedmiotów, które mają zostać włożone do plecaka. Zatem długość chromosomu będzie równa liczbie wszystkich przedmiotów n. Pojedynczy gen będzie definiował, czy dany przedmiot znajduje się w plecaku (wartość genu 1) czy też danego przedmiotu w tym plecaku nie ma (wartość genu 0). Przykładowy chromosom, dla zadania w którym występuje 15 przedmiotów, prezentuje diagram

nr przedmiotu	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
chromosom	1	1	0	0	0	1	0	1	0	0	0	0	0	1	0

poniżej. Chromosom ten koduje informację, o tym że w plecaku znajdują się przedmioty: $1, 2, 6, 8, 14$.

Funkcja przystosowania dla dyskretnego problemu plecakowego może być zdefiniowana w sposób w pełni zgodny z treścią zadania. Tzn. funkcja ta może rozróżnić dwa przypadki i: zwrócić sumę wartości wybranych przedmiotów, jeżeli suma ich objętości nie przekracza C (objętości plecaka); w przeciwnym razie zwrócić 0. Matematycznie można zapisać ją jako:

$$f(B) = \begin{cases} \sum_{(v_i,c_i)\in B} v_i, & \sum_{(v_i,c_i)\in B} c_i \leqslant C; \\ 0, & \sum_{(v_i,c_i)\in B} c_i > C, \end{cases} \tag{7.6}$$

gdzie dla czytelności i zgodności z definicją (7.5) argument B oznacza pewnego pojedynczego osobnika w naszej populacji.

Co jeśli wszystkie osobniki osiągną wartość przystosowania 0? Może się tak zdarzyć dla odpowiednio małej stałej C, gdy każdy osobnik ją przekroczy (choćby nieznacznie). Nie jest to sytuacja przyjemna szczególnie z punktu widzenia implementacji. Potencjalnych sposobów wybrnięcia z niej jest kilka: wygaszenie losowo wybranych jedynek z dużej części populacji, wykonanie selekcji ruletkowej zgodnie z rozkładem jednostajnym tzn. z prawdopodobieństwami $1/m$ dla każdego osobnika (selekcje rankingowa i turniejowa zwykle mogą być wykonane bez zmian), wykonanie dodatkowej funkcji przystosowania przypisującej większe wartości osobnikom słabiej naruszającym ograniczenie C (specjalna funkcja przeznaczona tylko dla tego przypadku).

Rozwiązanie dokładne

Dyskretny problem plecakowy może zostać rozwiązany w sposób dokładny za pomocą programowania dynamicznego. Potrzebujemy zdefiniować odpowiednią wielkość indukcyjną (wraz z krokiem indukcyjnym), a następnie napełnić pewną tablicę rozwiązaniami postępując od problemów „małych” do „dużych”.

Zdefiniujmy wielkość $V_{i,j}$ oznaczającą: wartość najlepszego upakowania plecaka o objętości j, za pomocą przedmiotów o numerach ze zbioru $\{1, \ldots, i\}$ (lub zbioru pustego dla $i = 0$). Po chwili zastanowienia się, można zorientować się, że wielkość ta powinna być określona następująco:

$$\begin{aligned} V_{0,j} &= 0, \\ V_{i,0} &= 0, \\ V_{i,j} &= V_{i-1,j}, && \text{jeżeli} \quad c_i > j, \\ V_{i,j} &= \max(V_{i-1,j},\, V_{i-1,j-c_i} + v_i), && \text{jeżeli} \quad c_i \leqslant j. \end{aligned} \tag{7.7}$$

Ostatnia linia reprezentuje zasadniczy krok indukcyjny.

Przekładając powyższy wzór na pseudokod otrzymujemy Algorytm 12. Podany algorytm oddaje na wyjściu tylko wartość najlepszego upakowania, ale nie określa, które przedmioty należy wybrać, aby osiągnąć tę wartość. W celu wskazania tego interesującego nas podzbioru przedmiotów należałoby wprowadzić pomocniczą tablicę tzw. wskazań wstecznych (ang. *back-pointers*) i na końcu ją prześledzić. Śledzenie wskazań wstecznych jest typowym etapem końcowym się w wielu podejściach opartych na programowaniu dynamicznym.

Algorytm 12 Algorytm rozwiązywania dyskretnego problemu plecakowego

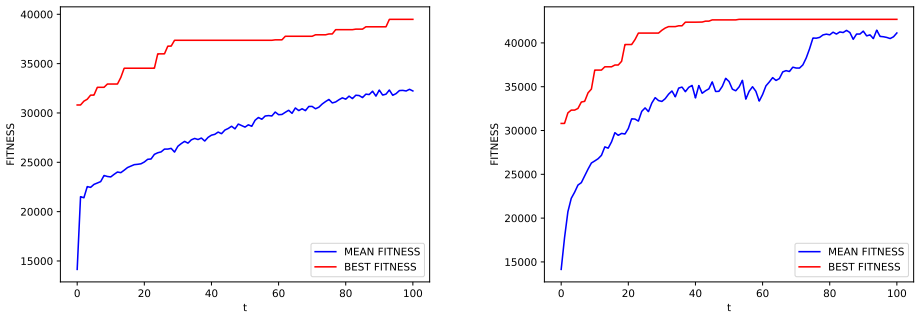
```
1:  procedura AlgorytmDyskretnyProblemPlecakowy
2:      dla i = 0,...,n wykonaj
3:          V_{i,0} := 0
4:      dla j = 1,...,C wykonaj
5:          V_{0,j} := 0
6:      dla j = 1,...,C wykonaj
7:          dla i = 1,...,n wykonaj          ▷ bierzemy pod uwagę i pierwszych przedmiotów
8:              jeżeli c_i > j to            ▷ sprawdzenie czy przedmiot mieści się
9:                  V_{i,j} := V_{i-1,j}     ▷ w plecaku o rozmiarze j
10:             w przeciwnym razie
11:                 V_{i,j} := max(V_{i-1,j}, V_{i-1,j-c_i} + v_i)
12:     zwróć V_{n,C}
```

Złożoność obliczeniowa zaprezentowanego algorytmu wynosi $\Theta(n\,C)$. Złożoność ta pozornie wydaje się liniowa, ponieważ n „wygląda” jak zmienna, a C „wygląda” jak stała. Tak naprawdę obie te wielkości są parametrami problemu i łatwo jest pomyśleć o ciągu problemów wejściowych o rosnących rozmiarach, gdzie C będzie skalowało się wraz z 2^n. Spowoduje to, że znalezienie rozwiązania będzie wymagało czasu wykładniczego. I faktycznie, ogólny dyskretny problem plecakowy jest zaliczany do klasy problemów NP-trudnych.

Na rys. 7.2 przedstawiono wykresy funkcji przystosowania (średnie i najlepsze) pochodzące z przykładowych wykonań dwóch algorytmów genetycznych na rzecz problemu plecakowego o rozmiarze $n = 100$. W obu przypadkach zadano: rozmiar populacji $m = 1000$, liczbę iteracji $T = 100$ oraz krzyżowanie dwupunktkwe, przy czym wykresy po lewej stronie dotyczą algorytmu stosującego selekcję ruletkową, zaś te po prawej algorytmu stosującego selekcję rankingową. Pierwszy algorytm zdołał znaleźć rozwiązanie stanowiące 92.46% rozwiązania dokładnego (ze względu na sumę wartości wybranych przedmiotów). Drugi algorytm znalazł rozwiązanie stanowiące 99.96% rozwiązania dokładnego.

AG z selekcją ruletkową
92.46% rozwiązania dokładnego

AG z selekcją rankingową
99.96% rozwiązania dokładnego



Rys. 7.2: Wykresy średniego i najlepszego przystosowania w kolejnych pokoleniach dla przykładowych wykonań algorytmów genetycznych dla problemu plecakowego o rozmiarze $n = 100$. Nastawy wspólne: $T = 100$, krzyżowanie dwupunktowe wykonywane z prawdopodobieństwem 0.9, prawdopodobieństwo mutacji na poziomie genu równe 10^{-3} (źródło: *opracowanie własne*).

7.2.2 Problem komiwojażera

Drugim klasycznym problemem, na przykładzie którego bardzo często tłumaczy się działanie algorytmu genetycznego, jest problem komiwojażera (ang. TSP — *Travelling Salesman Problem*). W problemie tym komiwojażer czyli obwoźnu sprzedawca ma za zadanie wyruszyć z miasta domowego, odwiedzić n miast (każde jednokrotnie) i powrócić do miasta domowego, i chciałby przy tym pokonać jak najkrótszą drogę. Podobnie jak DKP, TSP również jest problemem NP-trudny.

W implementacji rozwiązania tego problemu należy posłużyć się chromosomami kodowanymi liczbami całkowitymi, w których wartości poszczególnych genów nie mogą się powtarzać w danym chromosomie. Każdy chromosom będzie w istocie permutacją liczb od 1 do n reprezentującą pewną ścieżkę komiwojażera, gdzie wartość każdego genu będzie stanowić numer miasta. W rozważaniach poniżej założymy, że miasto domowe komiwojażera — miasto, z którego wyrusza, i w którym kończy podróż — ma numer 0 i nie jest jawnie zapisane w chromosomie, ale zakładamy jego obecność na krańcach jak w poniższym przykładzie dla $n = 14$.

0	5	2	11	8	4	3	10	14	6	7	13	9	12	1	0

A zatem każdy chromosom definiuje kolejność, w jakiej komiwojażer odwiedzałby miasta. Niech d_{ij} reprezentuje odległość (lub ogólniej koszt podróży) pomiędzy miastami i oraz j. Zwyczajowo zestaw takich odległości jest podany na

wejście problemu, np. jako wagi krawędzi grafu opisującego TSP. Na potrzeby algorytmu genetycznego wartość funkcji przystosowania pewnego osobnika x (dla prostoty pomijamy numer osobnika w ramach populacji) możemy określić jako sumę odległości poszczególnych połączeń i zapisać jako:

$$f(x) = d_{0,x_1} + \sum_{k=1}^{n-1} d_{x_k,x_{k+1}} + d_{x_n,0}. \tag{7.8}$$

Jednakże problem postawiony w ten sposób przybiera postać zadania minimalizacji, a w klasycznym algorytmie genetycznym zwyczajowo oczekuje się (ze względu na sposób działania etapu selekcji), że postawione zostanie mu zadanie maksymalizacji. Istnieje kilka sposób przekształcenia funkcji (7.8) na postać odpowiednią dla zadania maksymalizacji. Jeżeli liczby d_{ij} reprezentujące koszty są nieujemne, to wystarczającym zabiegiem będzie ozdobienie minusem całej sumy ze wzoru (7.8), tj.:

$$f(x) = -\left(d_{0,x_1} + \sum_{k=1}^{n-1} d_{x_k,x_{k+1}} + d_{x_n,0} \right). \tag{7.9}$$

Innym sposobem jest unormowanie wszystkich przystosowań obecnych w populacji do przedziału $[0,1]$, w taki sposób, aby koszt najmniejszy odwzorował się w wartość przystosowania 1, a największy w wartość przystosowania 0. Przy czym zabieg unormowania należy powtarzać w każdej iteracji algorytmu ze względu na zmieniająca się zawartość populacji i tym samym koszty dróg, które pokonują osobniki w niej zawarte. Otrzymany wówczas wzór można zapisać w postaci

$$f(x) = \frac{d_{0,x_1} + \sum_{k=1}^{n-1} d_{x_k,x_{k+1}} + d_{x_n,0} - f_{\max}}{f_{\min} - f_{\max}}, \tag{7.10}$$

gdzie $f_{\min}$, $f_{\max}$ to odpowiednio najmniejsza i największa wartość w ramach aktualnej populacji wyznaczona tymczasowo wg prawej strony wzoru (7.8).

W przypadku problemu komiwojażera należy stosować odpowiednie operatory genetyczne, takie jak np. krzyżowania z częściowym odwzorowaniem i mutację przez inwersję.

7.3 Ćwiczenia laboratoryjne (MATLAB)

E **Ćwiczenie 7.1 Napisz skrypt rozwiązujący dyskretny problem plecakowy za pomocą algorytmu genetycznego.** Polecenia do wykonania:

- Napisz ogólny skrypt realizujący kroki algorytmu genetycznego. Parametrami dla skryptu powinny być m.in. rozmiar populacji, liczba iteracji, wskaźnik na funkcję przystosowania, wskaźnik na funkcję selekcji, wskaźnik na funkcję krzyżowania, wskaźnik na funkcję mutacji.

- Napisz skrypt losujący problem plecakowy dla zadanej liczby przedmiotów — n.
- Napisz funkcję obliczającą wartości funkcji przystosowania dla podanego w parametrze chromosomu reprezentującego problem plecakowy.
- Napisz funkcje realizujące: selekcję ruletkową, krzyżowanie jednopunktowe, mutację (wszystkie te funkcje powinny przyjmować na wejście całą populację).
- Dla wylosowanego problemu plecakowego przeprowadź działanie algorytmu genetycznego. W każdej iteracji odnotuj, a następnie przedstaw na wykresie:
  1. średnie przystosowanie populacji,
  2. przystosowanie najlepszego osobnika w danym pokoleniu,
  3. przystosowanie najlepszego osobnika wykrytego w dotychczasowej historii.

Ćwiczenie 7.2 Napisz funkcje realizujące selekcje rankingową i turniejową oraz krzyżowanie dwupunktowe (wykorzystaj program z Ćwiczenia 7.1). Polecenia do wykonania:

- Napisz dwie funkcje selekcyjne realizujące: selekcję rankingową i turniejową.
- Porównaj działanie selekcji koła ruletki, rankingowej i turniejowej.
- Napisz funkcję do krzyżowania dwupunktowego.
- Dla wylosowanego problemu plecakowego przeprowadź działanie algorytmu genetycznego. Przeprowadź eksperymenty numeryczne porównujące działanie poszczególnych metod selekcji (koła ruletki, turniejowej, rankingowej) oraz poszczególnych metod krzyżowania (1-PX, 2-PX). W każdej iteracji odnotuj, a następnie przedstaw na wykresie:
  1. średnie przystosowanie populacji,
  2. przystosowanie najlepszego osobnika w danym pokoleniu,
  3. przystosowanie najlepszego osobnika wykrytego w dotychczasowej historii.

Ćwiczenie 7.3 Porównaj rozwiązanie dyskretnego problemu plecakowego przez algorytm genetyczny z rozwiązaniem dokładnym. Napisz skrypt rozwiązujący problem plecakowy w sposób dokładny. Bazując na programie napisanym do Ćwiczenia 7.1 dla wylosowanego problemu plecakowego (lub kilku) sprawdź, czy algorytm genetyczny zwraca ten sam wynik, co rozwiązanie dokładne.

Ćwiczenie 7.4 Napisz skrypt rozwiązujący problem komiwojażera za pomocą algorytmu genetycznego (wykorzystaj program z Ćwiczenia 7.1 oraz 7.2). Polecenia do wykonania:

- Napisz skrypt do losowania problemu komiwojażera dla zadanej liczby miast — n.
- Napisz funkcję obliczającą wartość funkcji przystosowania dla podanego w parametrze chromosomu reprezentującego problem komiwojażera.
- Zaimplementuj operatory krzyżowania PMX, OX, CX oraz odpowiednią mutację dla rozwiązywanego problemu.

– Dla wylosowanego zestawu miast przeprowadź działanie algorytmu genetycznego. Przeprowadź eksperymenty numeryczne porównujące działanie poszczególnych metod krzyżowania (PMX, OX, CX). W każdej iteracji odnotuj, a następnie przedstaw na wykresie:
  1. średnie przystosowanie populacji,
  2. przystosowanie najlepszego osobnika w danym pokoleniu,
  3. przystosowanie najlepszego osobnika wykrytego w dotychczasowej historii.

IV Systemy z wiedzą

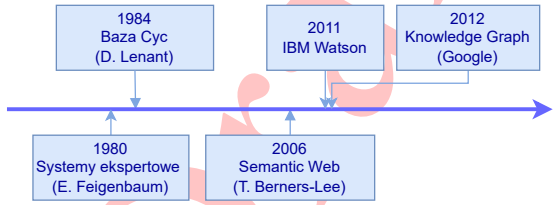
8. Systemy z wiedzą — wprowadzenie

W sztucznej inteligencji od momentu powołania tej dziedziny, czyli latach 50-tych poprzedniego wieku, systemy oparte na wiedzy (ang. KBS — *Knowledge Based Systems*) pozwalają komputerom / programom naśladować ludzką inteligencję a w szczególności sposób rozumowania. Efektywne przetwarzanie i wykorzystywanie wiedzy pozwala systemom na podejmowanie decyzji, rozwiązywanie złożonych zadań, takich jak wnioskowanie, uczenie się, planowanie i „rozumienie" środowiska, z którym współdziałają. Głównym trzonem systemów z wiedzą jest jej sformalizowana postać. Wiedza najczęściej pochodzi od specjalistów, dla których rozwiązywany problem leży w zakresie ich ekspertyzy. Właśnie ta cecha, czyli pozyskiwanie wiedzy od człowieka, odróżnienia systemy z wiedzą od modeli, które powstają przy wykorzystaniu metod maszynowego uczenia się, takich jak omawiane wcześniej sieci neuronowe, czy modele probabilistyczne, gdzie z danych uzyskuje się wiedzę i wyraża w parametrach modelu.

Najczęściej KBS są związane z przetwarzaniem symboli. Wczesne systemy symboliczne oparte były na systemach matematycznych. Przykładem takiego podejścia jest „Logic Theorist", który potrafił udowodnić 38 z pierwszych 52 twierdzeń podanych przez Bertranda Russella w Principia Mathematica. Innym osiągnięciem z tego okresu jest język programowania LISP (ang. *list processing*) opracowany przez John'a McCarthy'ego. Kolejnym etapem rozwoju w przetwarzaniu symbolicznym

były systemy ekspertowe, które mają praktyczne wykorzystanie poza obszarem matematyki i programowania, na przykład w biznesie i medycynie.

Kamienie milowe dla systemów KBS pokazuje diagram z rysunku 8.1. W latach 80-tych Edward Feigenbaum wprowadza termin „knowledge engineering", wskazując na znaczenie pozyskiwania wiedzy, reprezentacji wiedzy i zastosowania wiedzy w inteligencji maszynowej. Zainspirowane inżynierią wiedzy, zaczęły pojawiać się projekty złożonych systemów ekspertowych: baza wiedzy i projekt o nazwie *Cyc* i jego rozwinięcie *Semantic Web*. Najbardziej znaczącym osiągnięciem systemów KBS jest system Watson opracowany przez IBM. IBM Watson pokonał dwóch ludzkich zawodników w teleturnieju Jeopardy! (polaska edycja nosi nazwę Va banque), demonstrując potencjalną skuteczność systemu z bogatą wiedzą [LLS20]. I ostatnim wskazanym przełomowym projektem jest „Knowledge Graph" firmy Google, zmieniający wyszukiwanie na kontekstowe. W systemie zgromadzono ponad 500 miliardów faktów o pięciu miliardach podmiotów.



Rys. 8.1: Rozwój systemów z wiedzą na przestrzeni lat (źródło:(LLS20)).

Najważniejszym zagadnieniem naukowym w systemach z wiedzą jest sposób wyrażania wiedzy (reprezentacja wiedzy) i efektywne jej przetwarzanie celem symulowania rozumowania. W dalszej części niniejszy rozdział jest poświęcony reprezentacji wiedzy, u podstaw której leżą logiki, między innymi logika predykatów pierwszego rzędu, która stanowi podstawę dla programowania logicznego, którego przykładem jest język Prolog. W języku tym definiuje się fakty i reguły do rozwiązywania problemów poprzez wnioskowanie logiczne. Logika dwuwartościowa, jaką jest logika predykatów pierwszego rzędu jest dość ograniczona w symulowaniu rozumowania ludzkiego, dlatego opracowano różne koncepcje radzenia sobie z niepewnością postrzegania świata. Jednym z podejść do niepewności jest logika rozmyta. Pozwala ona na obsługę koncepcji częściowej prawdy — gdzie wartość prawdy może wahać się między całkowicie prawdziwą a całkowicie fałszywą. Inaczej reprezentują niepewność świata systemy probabilistyczne takie jak sieci bayesowskie, które dostarczają środków do modelowania probabilistycznych zależności między zestawem zmiennych losowych.

8.1 Definicja wiedzy

Wykorzystanie systemów z wiedzą wymaga co najmniej próby określenia, czym jest *wiedza*. Można ją zdefiniować jako zbiór informacji, umiejętności i przekonań nabytych przez doświadczenie, edukację lub odkrywanie, które mogą być wykorzystane do rozumienia, przewidywania lub wyjaśniania zjawisk. W różnych kontekstach rozumienie pojęcia *wiedza* skupia się na innch jej aspektach. I tak:

1. **Filozofia:** Wielki myśliciel Sokrates stwierdził, że istotę prawdziwej wiedzy stanowią pojęcia, które mają swoje odzwierciedlenie w realnym świecie. Natomiast, według Platona wiedza jest postrzegana jako „uzasadnione przekonanie zgodne z rzeczywistością".
2. **Psychologia poznawcza:** opisuje wiedzę jako informacje przechowywane w pamięci, które są organizowane i przechowywane w sieciach semantycznych (1975 r.) [CL75].
3. **Zarządzanie wiedzą:** Nonaka i Takeuchi definiują wiedzę jako proces tworzenia znaczeń przez interakcję społeczną. Jest to strategiczny zasób i kluczowy element intelektualnego kapitału organizacji (1995 r.) .
4. **W** informatyce wiedza to dane przetworzone przez algorytmy, z reguły do formy umożliwiającej systemom podejmowanie decyzji (2004 r.) [BL04].
5. **W** systemach ekspertowych wiedza jest zbiorem reguł i heurystyk do symulacji rozumowania eksperta (1998 r.) [Nil98].

8.2 Reprezentacja

Rozumiejąc już czym jest wiedza możemy posłużyć się przykładem z pewnego obszaru uznając, że mamy wystarczający poziom ekspertyzy, by odpowiedzieć na pewne pytanie. Znalezienie na nie odpowiedzi jest związane z posiadaniem wiedzy. Postawmy zatem pytanie: „Czym jest Wiedźmin?". Odpowiedzi mogą być różne, bo zależą od punktu widzenia. I tak, „Wiedźmin" to cykl powieści Andrzeja Sapkowskiego, ale może to być też gra komputerowa, lub bohater książki, mutant wyszkolony by walczyć z potworami. Każda z odpowiedzi wyrażona w formie swobodnej w języku polskim będzie trudna do przedstawienia w systemie komputerowym. Zatem w KBS należy posłużyć się uproszczoną formą nazywaną reprezentacją [Ber18].

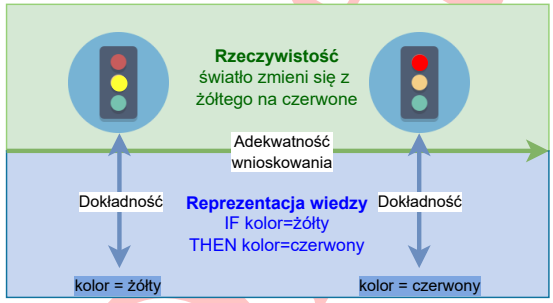
Reprezentacja wiedzy stanowi fundamentalne pojęcie w sztucznej inteligencji, ponieważ umożliwia systemom AI rozumienie i przetwarzanie informacji. Jej celem jest, aby wiedzę ludzką przekształcić w formę, którą komputery mogą przetwarzać na potrzeby symulowania inteligentnego zachowania. Dobry system reprezentacji wiedzy musi posiadać dwie właściwości:

1. **Dokładność reprezentacji**: pozwala reprezentować dowolny koncept, relację

i obserwację ze świata rzeczywistego.

2. **Adekwatność wnioskowania**: jest w stanie manipulować strukturami w celu wytworzenia nowej wiedzy w taki sposób, by powstały wynik odpowiadał temu, co wydedukowałby człowiek.

Rys. 8.2 przedstawia przykład dokładności i adekwatności reprezentacji. W bardzo uproszczonych warunkach, aby wskazać, że świeci się żółte światło drogowe wystarczy przypisać zmiennej `kolor` wartość `żółty`. Kierowca w takiej sytuacji będzie oczekiwał, że jako kolejne zapali się światło czerwone. Odwzorowanie tego sposobu rozumowania można przedstawić w postaci prostej reguły warunkowej, że `jeżeli kolor = żółty to następnie kolor = czerwony`. Jak widać w kolejnym kroku zarówno w świecie rzeczywistym jak i w reprezentacji wiedzy uzyskujemy spójny i tożsamy stan.



Rys. 8.2: Przykład pokazujący dokładność i adekwatność reprezentacji na przykładzie wiedzy o zmieniających się światłach drogowych (źródło: opracowanie własne na podstawie (RN09)).

Ostatnim zagadnieniem w tym podrozdziale będzie omówienie celu reprezentacji wiedzy. Katalog zadań, które można realizować z wykorzystaniem systemów opartych na wiedzy, które wykorzystują pewną formę reprezentacji można ująć w postaci następującej listy:

1. **Wnioskowanie**: Wykorzystanie wiedzy do generowania nowych faktów i wyciągania logicznych wniosków na podstawie dostępnych informacji. Przykładem może być system ekspertowy do diagnozowania chorób, taki jak MYCIN, który używał reguł wnioskowania do określania źródła i rodzaju zakażenia bakteryjnego i rekomendowania odpowiedniej antybiotykoterapii.
2. **Rozumienie języka naturalnego**: Interpretacja i generowanie ludzkiego języka w sposób, który umożliwia komunikację z użytkownikami na wysokim poziomie abstrakcji. Przykładem może być SHRDLU - klasyczny program komputerowy do przetwarzania języka naturalnego, który wykorzystuje bazę wiedzy o świecie złożonym z bloków do interpretacji i reagowania

na komendy językowe.

3. **Przetwarzanie obrazów i wideo**: Analiza i interpretacja treści wizualnych, aby rozpoznać obiekty, sytuacje i zachowania w obrazach i sekwencjach wideo. Przykładem mogą być systemy rozpoznające znaki drogowe, które wykorzystują bazy wiedzy składające się z zestawów danych z obrazami i opisami znaków drogowych. Przetwarzają one symboliczne reprezentacje tych znaków, które są następnie używane do identyfikacji i klasyfikacji w czasie rzeczywistym przez aplikacje nawigacyjne lub systemy wspomagające kierowcę.
4. **Automatyzacja i robotyka**: Wykorzystanie wiedzy do nawigacji, manipulacji i interakcji robotów z fizycznym światem. Na przykład reguły określające użyteczność elementu, na podstawie których robot wykonujący kontrolę jakości podejmie decyzję o jego ewentualnym odrzuceniu do odpadów.
5. **Planowanie**: Tworzenie i wykonanie sekwencji działań w celu osiągnięcia określonego celu. Przykładem mogą być systemy planowania misji kosmicznych, które tworzą szczegółowe sekwencje działań dla sond kosmicznych i łazików marsjańskich.

Podsumowując, systemy oparte na wiedzy rozwijają się od ponad 70 lat i w chwili obecnej w niewielkim stopniu spotyka się „czyste” systemy KBS, natomiast często łączy się je z innymi podejściami takimi jak modelowanie danych, optymalizacja czy inne podejścia do podejmowania decyzji.

9. Logika pierwszego rzędu

W sztucznej inteligencji od 1958 roku stosowano logikę do przetwarzania informacji zawartych w deklaratywnych bazach wiedzy. Właśnie wtedy John McCarthy zalecił logikę do prezentowania informacji o specyficznej dziedzinie problemu i opisał charakterystykę programowania deklaratywnego, często analogicznego do języka logiki predykatów pierwszego rzędu. Ponadto stosował reguły logiki do wyciągania wniosków z zapisanej wiedzy.

Jeżeli logika jest używana do rozwiązania „rzeczywistych" problemów, to wymaga eksperckiej znajomości danej dziedziny.

Definicja 9.0.1 Logika to dyscyplina filozoficzna i matematyczna zajmująca się formalnymi zasadami wnioskowania. Zajmuje się ona strukturami argumentacji, aby ocenić, czy dane rozumowanie jest poprawne. Logika dostarcza narzędzi umożliwiających formalizację i ocenę prawdziwości zdań oraz relacji między nimi.

Własności logiki pierwszego rzędu zostały zestawione w tabeli 9.1. Własności te przedstawiono syntetycznie i pełne ich zrozumienie wymagałoby wprowadzenie szerokiego wachlarza pojęć i definicji, które wykraczają poza plan niniejszego wywodu. Dalsza część skryptu skupia się na praktycznym wykorzystaniu logiki pierwszego rzędu, zetem tylko część z własności zostanie omówiona w szczegółach.

Tabela 9.1: Własności rachunku predykatów pierwszego rzędu.

Nr	Nazwa własności	Ogólny opis	Logika predykatów I rzędu
1	Spójność (ang. consistency)	System logiczny nie prowadzi do sprzecznych wniosków.	Możliwe sprzeczności wynikają tylko z niespójnych aksjomatów; system jest spójny, jeśli jego aksjomaty są spójne.
2	Zupełność (ang. completeness)	System jest zupełny, jeśli można w nim udowodnić każde prawdziwe zdanie.	FOL jest zupełny w sensie teoretycznym, zgodnie z twierdzeniem Gödla o zupełności.
3	Poprawność (ang. soundness)	System jest poprawny, jeśli każde udowodnione w nim zdanie jest prawdziwe.	Każde zdanie udowodnione w logice pierwszego rzędu jest prawdziwe w każdym jego modelu.
4	Rozstrzygalność (and. decidability)	Możliwość ustalenia prawdziwości każdego zdania za pomocą skończonego procesu.	Nie wszystkie zdania są rozstrzygalne ze względu na nieskończoną liczbę potencjalnych interpretacji.
5	Ekspresyjność (ang. expressiveness)	Zdolność wyrażania bogatego zestawu koncepcji.	Pozwala na wyrażanie szerokiego zakresu zdań dotyczących obiektów i ich relacji.
6	Dedukcyjność (ang. deductibility)	Zdolność do wyprowadzania wniosków na podstawie dostępnych informacji i reguł.	Umożliwia dedukcję bardziej złożonych struktur dzięki kwantyfikatorom i zmiennym.

9.1 Składnia i semantyka

Każda logika może być rozważana w kontekście dwóch komponentów: składni i semantyki (znaczenia).

Rachunek predykatów pierwszego rzędu inaczej logika pierwszego rzędu (ang. First Order Logic, FOL), jest systemem logicznym, który służy do analizowania struktury i znaczenia zdań dotyczących obiektów i ich relacji. Logika predykatów jest logiką dwuwartościową, czyli każda formuła jest albo prawdziwa, przyjmuje wartość TRUE lub jest fałszywa, przyjmuje wartość FALSE.

9.1.1 Składnia

Składnia rachunku predykatów pierwszego rzędu określa formalne reguły budowania poprawnych wyrażeń w tym systemie. Wyrażenia zawierają zmienne, stałe, funkcje, predykaty, kwantyfikatory i łączniki logiczne, które są skonstruowane zgodnie z określonymi zasadami gramatycznymi.

Poniższa lista przedstawia podstawowe elementy składniowe w logice pierw-

szego rzędu oraz przyjętą konwencję ich oznaczania, tj.:

- *Symbole*:
  - Zmienne: *x, y, z* : to ciągi znaków alfanumerycznych zaczynające się od małej litery.
  - Stałe: *Geralt , Proba_Traw, Nilfgard, 15*. to ciągi znaków alfanumerycznych zaczynające się od wielkiej litery lub cyfry.
  - Funkcje: *opiekun[2], znak[2], ryzyko[1]*. Nazwy funkcji to ciągi znaków alfanumerycznych zaczynające się zawsze od małej litery a wartość w klamrach to ich wymiarowość (liczba argumentów). Mają one taką cechę, że zwracają wartość.
  - Predykaty: *Kocha[2], Sasiad[2], Driada[1]*: Nazwy predykatów składać się będą ze znaków alfanumerycznych i zaczynają się wielką literą a wartość w klamrach to ich wymiarowość (liczba argumentów). Predykaty reprezentują pewną relację, która wiąże jej argumenty.
  - Kwantyfikatory: ogólny (duży, uniwersalny) (czytaj: dla każdego) $\forall$ i szczególny (mały, egzystencjalny) (czytaj: istnieje) $\exists$.
  - Łączniki logiczne: koniunkcja $\wedge$, dysjunkcja $\vee$, negacja $\neg$, implikacja $\rightarrow$, równoważność $\leftrightarrow$.
  - Symbol równości termów: =. Termy, inaczej wyrażenia logiczne to zbiór zawierający: { stałe, zmienne i funkcje}.
  - Pomocnicze: takie jak nawiasy okrągłe () i przecinki używane do organizowania wyrażeń.
- *Formuły*: to większe struktury niż symbole. Składnia określa zasady jak elementy mogą być łączone w formuły, czyli wyrażenia:
  - Formuła atomowa: Driada(x), gdzie Driada jest symbolem predykatu, a x jest zmienną.
  - Negacja: $\neg$Driada(x);
  - Koniunkcja: Driada$(x) \wedge$ Elf(x);
  - Dysjunkcja: Driada$(x) \vee$ Elf(x);
  - Implikacja: Driada$(x) \rightarrow$ Elf(x);
  - Równoważność: Driada$(x) \leftrightarrow$ Elf(x);
  - Kwantyfikacja uniwersalna: $\forall x$(Driada(x));
  - Kwantyfikacja egzystencjalna: $\exists x$(Elf(x));
  - Równość termów: Protektor(Ciri) $=$ Geralt.

Przykładowa poprawna składniowo formuła w rachunku predykatów pierwszego rzędu:

$$\forall x(\text{Elf}(x) \rightarrow \exists y(\text{Czlowiek}(y) \wedge \text{Nienawidzi}(x,y))) \tag{9.1}$$

Przykładowa niepoprawna składniowo formuła w rachunku predykatów pierw-

szego rzędu:

$$\leftrightarrow (\text{Elf}(\forall)\exists y(\text{Czlowiek}(y) \wedge (x,y)))$$

Błędami jest brak lewej strony równoważności, użycie kwantyfikatora wielkiego bez wskazania zmiennej pod nim, predykat Elf ma niepoprawny argument, a zmienne (x,y) nie są związane żadną relacją.

9.1.2 Semantyka

Zastanówmy się nad znaczeniem formuły 9.1. Oznacza, że dla każdego x, jeśli $\text{Elf}(x)$ jest prawdziwe, to istnieje taki y, że $\text{Czlowiek}(y)$ jest prawdziwe i Nienawidzi jest relacją między x i y. Potocznie można zdanie sformułować jako: „Każdy elf nienawidzi jakiegoś człowieka".

Interpretacja, czyli przypisanie znaczenia, naturalnie przenosi nas w obszar semantyki. Semantyka opisuje jakie wartości czyli znaczenie przypisuje się symbolom w formułach składniowych, określając, kiedy formuła jest prawdziwa. Każda formuła logiczna, jest oceniana w kontekście określonej interpretacji, która przypisuje:

1. Obiekty do stałych: Każdej stałej w formule przypisuje się konkretny obiekt z dziedziny (domeny).
2. Funkcje do symboli funkcyjnych: Symbole funkcyjne są interpretowane jako konkretne funkcje, które przypisują obiekty do obiektów w dziedzinie.
3. Relacje do symboli predykatów: Symbole predykatów są interpretowane jako specyficzne relacje między obiektami.

Kwantyfikatory są interpretowane jako odnoszące się do wszystkich obiektów w opisywanej dziedzinie (domenie) (dla kwantyfikatora uniwersalnego) lub do istnienia przynajmniej jednego obiektu (dla kwantyfikatora egzystencjalnego), który spełnia daną formułę.

Rozważmy przykład zdania z kwantyfikatorem uniwersalnym: „Każdy wiedźmin jest mutantem", co zapisujemy w logice predykatów:

$$\forall x(\text{Wiedzmin}(x) \rightarrow \text{Mutant}(x)).$$

Przyjmijmy, że $x = \text{Vesemir}$ lub $x = \text{Ciri}$ lub $x =$ bazyliszek, to tylko pierwszy Vesemir sprawia, że lewa strona implikacji jest prawdziwa, co prowadzi do prawdziwości prawej strony. Pamiętać należy, że całe wyrażenie jako implikacja jest fałszywe tylko w jednym przypadku, gdy z prawdy wynikałby fałsz.

Rozważmy przykład zdania z kwantyfikatorem egzystencjalnym: „Geralt jest kochany przez czarodziejkę" co zapisujemy w logice predykatów:

$$\exists x(\text{Czarodziejka}(x) \wedge \text{Kocha}(x, \text{Geralt})).$$

Przyjmijmy, że $x =$ Yennefer lub $x =$ Bazyliszek, to zdanie jest prawdziwe dla Yennefer, bo tylko ona sprawia, że oba predykaty w formule są prawdziwe.

Kwantyfikator uniwersalny łączy się tylko z implikacją, a kwantyfikator egzystencjalny tylko z koniunkcją.

Semantyka określa pojęcie *modelu* dla formuł. Model to interpretacja, w której formuła jest prawdziwa. Mówi się, że formuła jest prawdziwa w modelu, lub że model spełnia formułę. Zagadnienie spełnialności jest fundamentalne w teorii modeli - dziedzinie matematyki zajmującej się badaniem relacji między składnią logiczną a jej interpretacjami. Model musi dostarczać informację niezbędną do określenia czy zdanie jest prawdziwe czy fałszywe.

9.2 Wybrane aksjomaty logiki predykatów pierwszego rzędu

Rozważanie dalszych przykładów i omówienie schematów wnioskowania wymaga znajomości własności wyrażeń w rachunku predykatów pierwszego rzędu. Wprowadźmy pojęcie tautologii.

Definicja 9.2.1 Tautologie logiki pierwszego rzędu są formułami prawdziwymi w dowolnym zbiorze (inaczej modelu). Prawdziwość zdań nie wynika z interpretacji znaczenia stałych, funkcji i predykatów, czy przypisywania zmiennym wartości, a jest konsekwencją tylko struktury wyrażenia.

Definicja 9.2.2 Mówimy, że dwie formuły / funkcje $A(x)$ i $B(x)$ (o wspólnym zakresie zmiennej x) są równoważne, gdy mają ten sam zakres / wykres. Innymi słowy, gdy X oznacza wspólny zakres zmiennej x:

$$\{x \in X : A(x)\} = \{x \in X : B(x)\}. \tag{9.2}$$

Definicja 9.2.3 Mówimy, że dwie formuły domknięte A i B są równoważne, gdy formuła $A \leftrightarrow B$ jest tautologią.

Definicja 9.2.4 Dwa predykaty $A(x)$ i $B(x)$, $x \in X$ są równoważne wtedy i tylko wtedy, gdy zdanie $\forall x(A(x) \leftrightarrow B(x))$ jest prawdziwe.

Poniża lista zawiera tautologie, które będą przydatne w rozumieniu przykładów i dalszych algorytmów.

Dla uproszczenia pojedyncze symbole A i B prezentują zdanie z rachunku zdań. Należy pamiętać, że wszystkie przestawione w ten sposób tautologie, mogą być uogólnione do formuł atomowych w logice predykatów pierwszego rzędu.

1. Tautologie wywodzące się z rachunku zdań:
   (a) Prawo wyłączonego środka (łac. *tertium non datur*): $A \vee (\neg A)$
   (b) Prawo sprzeczności (niemożliwe jest by formuła była jednocześnie prawdziwa i fałszywa): $\neg(A \wedge (\neg A))$
   (c) Prawo podwójnej negacji: $A \leftrightarrow \neg(\neg A)$
   (d) I prawo de Morgana: $\neg(A \wedge B) \leftrightarrow ((\neg A) \vee (\neg B))$
   (e) II prawo de Morgana: $\neg(A \vee B) \leftrightarrow ((\neg A) \wedge (\neg B))$
   (f) Prawo przemienności koniunkcji: $(A \wedge B) \leftrightarrow (B \wedge A)$
   (g) Prawo przemienności dysjunkcji:$(A \vee B) \leftrightarrow (B \vee A)$
   (h) Prawo łączności koniunkcji: $(A \wedge (B \wedge C)) \leftrightarrow ((A \wedge B) \wedge C))$
   (i) Prawo łączności dysjunkcji: $(A \vee (B \vee C)) \leftrightarrow ((A \vee B) \vee C))$
   (j) Prawo eliminacji równoważności: $(A \leftrightarrow B) \leftrightarrow (A \rightarrow B) \wedge (B \rightarrow A)$
   (k) Prawo eliminacji implikacji: $(A \rightarrow B) \leftrightarrow (\neg A \vee B)$
   (l) Rozdzielność koniunkcji względem dysjunkcji $(A \wedge (B \vee C)) \leftrightarrow ((A \wedge B) \vee (A \wedge C))$
   (m) Rozdzielność dysjunkcji względem koniunkcji $(A \vee (B \wedge C)) \leftrightarrow ((A \vee B) \wedge (A \vee C))$
2. Tautologie właściwe dla rachunku predykatów (kwantyfikatorów):
   (a) Prawa de Morgana rachunku kwantyfikatorów:
      i. $\neg\forall x A(x) \leftrightarrow \exists x \neg A(x)$
      ii. $\neg\exists x A(x) \leftrightarrow \forall x \neg A(x)$
   (b) Przemienność kwantyfikatorów ogólnych:
   $\forall x \forall y A(x,y) \leftrightarrow \forall y \forall x A(x,y)$
   (c) Przemienność kwantyfikatorów szczególnych:
   $\exists x \exists y A(x,y) \leftrightarrow \exists y \exists x A(x,y)$

Różne kwantyfikatory nie są przemienne. Rozważmy znaczenie zapisów, które na przykładzie pokażą, że interpretacja jest inna, gdy zmienimy kolejnością kwantyfikator uniwersalny z egzystencjalnym:

1. $\forall x(\exists y(Kocha(x,y)))$
2. $\exists y(\forall x(Kocha(x,y)))$

Pierwsze zdanie opisuje że każdy kogoś kocha, natomiast drugie wyraża, że istnieje ktoś (y), który jest przez wszystkich (x) kochany.

9.2.1 Założenie o unikalności nazw, zamkniętej dziedzinie i zamkniętym świecie

Aby wyrazić w logice predykatów że Geralt, Vesemir i Eskel są wiedźminami możemy wprowadzić stałe Geralt, Vesemir i Eskel i jednoargumentowy predykat Wiedzmin, a następnie użyć zdań atomowych

Wiedzmin(Geralt),
Wiedzmin(Vesemir),
Wiedzmin(Eskel).

Wszystkie trzy obiekty mapują na ten sam element opisywanej książkowej rzeczywistości. Wszystkie stałe reprezentujące imiona powinny zatem spełniać warunek: $\text{Geralt} \neq \text{Vesemir}, \text{Geralt} \neq \text{Eskel}, \text{Vesemir} \neq \text{Eskel}$, gdyż odnoszą się do różnych obiektów.

Własność 9.2.1 Założenie o unikalności nazw dla predykatu lub funkcji jest wyrażone przez formuły:

$$\forall x_1 \ldots x_m y_1 \ldots y_n (f(x_1, \ldots, x_m) \neq g(y_1, \ldots, y_n)) \tag{9.3}$$

dla wszystkich par funkcji f, g, oraz

$$\forall x_1 \ldots x_n y_1 \ldots y_n (f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n) \rightarrow (x_1 = y_1 \wedge \ldots \wedge x_n = y_n)) \tag{9.4}$$

dla wszystkich stałych funkcji f o liczbie argumentów większej niż 0. Te formuły pociągają za sobą $t_1 = t_2$ dla jakichkolwiek różnych termów t_1, t_2.

Własność wprowadzona przez Keitha Clarka w 1978 [HLP08] jest często używana w teorii programowania logicznego, nazywana założeniem o zamkniętej dziedzinie, które zakłada, że żaden model nie zawiera elementów domenowych innych, niż te wykorzystane w postaci symboli stałych.

Własność 9.2.2 Załóżmy, że używamy symboli predykatów lub funkcji zawierających skończoną liczbę argumentów, które są stałymi $C_1, \ldots, C_n$. Warunek zamkniętej dziedziny (ang. *Domain Closure Assumption*) opisuje wzór:

$$\forall x (x = C_1 \vee \cdots \vee C_n) \tag{9.5}$$

Warunek umożliwia zastąpienie wszystkich kwantyfikatorów w dowolnej formule wielokrotnymi koniunkcjami (np. $\forall x(P(x) \leftrightarrow P(C_1) \wedge \cdots \wedge P(C_n))$) i dysjunkcjami (np. $\exists x(P(x) \leftrightarrow P(C_1) \vee \cdots \vee P(C_n))$).

W dziedzinie programowania logicznego wykorzystuje się trzecie założenie o zamkniętym świecie (ang. Closed World Assumption), które zakłada, że zdania atomowe, o których nie wiemy, że są prawdziwe (nie są jawnie wskazane jako prawdziwe) uważa się za fałszywe. Ma to istotne znaczenie przy projektowaniu baz wiedzy, których status uznaje się zawsze za TRUE. Wszystko zatem co w bazie wiedzy nie jest opisane jest FALSE.

9.3 Inżynieria wiedzy

Inżynieria wiedzy to proces modelowania problemu za pomocą formalnych metod, takich jak na przykład logika predykatów, oraz do dalszego wykorzystania tego modelu do wnioskowania i rozwiązywania konkretnych problemów. Lista kroków to:

1. **Identyfikacja zadania:** To określenie celu lub problemu, który ma być rozwiązany. Wymaga zrozumienia kluczowych aspektów zadania oraz jego kontekstu. Wiąże się też z określeniem przebiegu dialogu, odkrycia jakie będą możliwe pytania do systemu.
2. **Zgromadzenie wymaganej wiedzy** Zebranie wszystkich niezbędnych informacji i danych, które są istotne dla zadania. Ta wiedza może pochodzić z różnych źródeł, w tym z literatury, ekspertów w danej dziedzinie, lub poprzez obserwacje.
3. **Wybór słownika predykatów, funkcji i stałych:** Określa się i nazywa predykaty, funkcje i stałe, które będą używane do reprezentowania wiedzy w ramach zadania. To pomoże w formalizacji problemu.
4. **Kodowanie ogólnej wiedzy dotyczącej zadania:** zapisanie wszystkie aksjomatów z wykorzystaniem termów ze słownika. Na tym etapie wykrywa się luki i błędy koncepcyjne.
5. **Kodowanie specyficznej wiedzy dotyczącej zadania:** Zakodowanie informacji szczegółowych i specyficznych, wejść do systemu lub informacji znanych podczas uruchomienia systemu. Inaczej określa się ją jako fakty.
6. **Stawianie pytań do procedury wnioskowania i interpretacja odpowiedzi.** Wykorzystanie systemu wnioskowania do zadawania pytań związanych z zadaniem i odebranie odpowiedzi, które pomogą w podejmowaniu decyzji lub dalszej analizie.
7. **Usuwanie błędów z bazy wiedzy:** Analiza wszystkich odpowiedzi systemu i poprawienie błędów lub niespójności w bazie wiedzy, aby zapewnić jej zgodność z oczekiwaniami projektanta.

Przeanalizujmy przykład pokazujący wykorzystanie logiki predykatów do opisu wiedzy czyli pewnych faktów i relacji oraz w dalszej części wnioskowania. Spełnia on założenia o unikalności nazw, zamkniętej dziedzinie i zamkniętym świecie.

Zgodnie z krokami inżynierii wiedzy:

1. **Identyfikacja zadania:** Po krótce chcemy przedstawić wiedzę o relacji Wiedźmina Geralta z potworami, które najczęściej zabija na zlecenie.
2. **Zgromadzenie wymaganej wiedzy** Korzystając ze swojej pamięci i przewodników o cyklu powieści zebrano istotne dane przedstawione w tabeli 9.2 w kolumnie „Zdanie opisujące wiedzę”.

3. **Wybór słownika predykatów, funkcji i stałych:** Predykaty: Wojownik, Mieszka, Potwor, Wiedzmin, Mord_na_zlecenie, Obojetny, Walka; Stałe: Geralt, Khaer Morhen, Bazyliszek; funkcje: brak.
4. **Kodowanie ogólnej wiedzy dotyczącej zadania:** przedstawione w tabeli 9.2 w kolumnie „Wyrażenie logiczne reprezentujące zdanie" w wierszu: 3,5,6,7
5. **Kodowanie specyficznej wiedzy dotyczącej zadania:** Przedstawione w tabeli 9.2 w kolumnie „Wyrażenie logiczne reprezentujące zdanie" w wierszu: 1,2,4,8

Tabela 9.2: Przykładowa baza wiedzy prezentująca przejście z opisu lingwistycznego na bazę wiedzy w logice pierwszego rzędu.

Lp.	**Zdanie opisujące wiedzę**	**Wyrażenie logiczne reprezentujące zdanie**
1	Geralt jest wojownikiem.	$\text{Wojownik}(\text{Geralt})$
2	Geralt mieszka w Kaer Morhen.	$\text{Mieszka}(\text{Geralt}, \text{Kaer Morhen})$
3	Każdy mieszkaniec Kaer Morhen jest wiedźminem.	$\forall x(\text{Mieszka}(x, \text{Kaer Morhen}) \rightarrow \text{Wiedzmin}(x))$
4	Bazyliszek jest potworem.	$\text{Potwor}(\text{Bazyliszek})$
5	Wiedźmini albo zabijają Bazyliszka dla zarobku lub jest im obojętny.	$\forall x(\text{Wiedzmin}(x) \rightarrow \text{Mord\_Na\_Zlecenie}(x, \text{Bazyliszek}) \vee \text{Obojetny}(x, \text{Bazyliszek}))$
6	Każdy bywa obojętny wobec czegoś.	$\forall x(\exists y(\text{Obojetny}(x, y))$
7	Wojownicy walczą z potworem, który nie jest im obojętny.	$\forall x \forall y(\text{Wojownik}(x) \wedge \text{Potwor}(y) \wedge \text{Walka}(x, y) \rightarrow \neg\text{Obojetny}(x, y))$
8	Geralt walczył z Bazyliszkiem.	$\text{Walka}(\textit{Geralt}, \textit{Bazyliszek})$

Przykład w tabeli 9.2 jest bazą wiedzy (ang. KB — *Knowledge Base*). Przedstawione wyrażenia logiczne, to tylko jedna z możliwości przejścia z opisu ogólnego na sformalizowany, w szczególności:

- pomija następstwo czasu, kolejność zdarzeń;
- nie wskazuje wyraźnie, że jest to opis nierzeczywistego świata;
- brak jest określenia zasięgu w zdaniu 6; czy dla każdego istnieje ktoś dla kogo jest się obojętnym? czy są to różne osoby? czy istnieje ktoś, wobec kogo wszyscy są obojętni?
- zdanie 7 można też przedstawić jako
  $\forall x, y(\text{Wojownik}(x) \wedge \text{Potwor}(y) \wedge \neg\text{Obojetny}(x, y) \rightarrow \text{Walka}(x, y))$
  gdyż nie jest jednoznaczne, co jest przyczyną a co skutkiem.

Kolejny etap w inżynierii wiedzy „wnioskowanie" jest przedmiotem wywodu

w kolejnym podrozdziale.

9.4 Wnioskowanie w logice predykatów pierwszego rzędu

W logice predykatów można stosować różne podejścia do wnioskowania. Celem niniejszego wywodu jest wprowadzenie do programowania w logice, dlatego skupimy się tylko na elementach procesu wnioskowania, które są z nim związane.

9.4.1 Unifikacja

Unifikacja w logice pierwszego rzędu polega na znalezieniu substytucji, która sprawia, że dwie formuły atomowe stają się identyczne.

Substytucja / unifikacja to odwzorowanie ze zmiennych na termy, które jest tożsamością na wszystkich ale skończenie wielu zmiennych. Ogólna reguła mówi, że stała unifikuje się ze stałą, która jest identyczna, zmienna unifikuje się ze stałymi i funkcjami, a formuły złożone muszą ulec dekompozycji i każdy z argumentów jest sprawdzany pod względem możliwości substytucji. Zbiór $\{x_1/t_1, x_2/t_2, ..., x_n/t_n\}$ przedstawia substytucję mapującą zmienną x_i na term t_i, dla $1 \leq i \leq n$.

Na przykład zapis, $P(x, f(x))\{x/g(y)\} = P(g(y), f(g(y)))$ oznacza, że $P(g(y), f(g(y)))$ jest instancją $P(x, f(x))$.

Poniżej podany jest algorytm unifikacji o wykładniczej złożoności czasowej, który jest wystarczający do wielu praktycznych zadań. Jeśli $expr_1$ i $expr_2$ to dwa termy, a α jest najbardziej ogólnym unifikatorem $expr_1$ i $expr_2$, to $expr_1/\alpha$ może być wykładnicze względem $expr_1$ i $expr_2$, więc konstruowanie $expr_1/\alpha$ jest z natury wykładnicze, chyba że użyte zostanie odpowiednie kodowanie termów; oznacza to reprezentowanie powtarzających się sub-termów tylko raz.

Rozpatrzmy dwa przykłady, jeden, gdy algorytm Unifikacji zwróci listę substytucji i drugi, gdy algorytm zwróci niepowodzenie.

■ **Przykład 9.1** Przykłady możliwej unifikacji

1. **Wyrażenia do Unifikacji:**
   (1) Zabija(w, x)
   (2) Zabija(Geralt, Strzyga)

   Proces Unifikacji:
   - W pierwszej formule x, w są zmiennymi.
   - W drugiej formule mamy konkretne byty: Geralt i Strzyga.
   - Unifikacja poprzez zastąpienie w na Geralt i x na Strzyga.

   Wynik Unifikacji: Substytucja: $\{w/\text{Geralt}, x/\text{Strzyga}\}$
   Po zastosowaniu substytucji, oba wyrażenia stają się identyczne: Zabija(Geralt, Strzyga)

Algorytm 13 Algorytm Unifikacja

```
1:  procedura UNIFIKACJA(Exp)                          ▷ Zbiór par wyrażeń do unifikacji. Expr
2:      dopóki Expr ≠ {} wykonaj                       ▷ istnieją pary do unifikacji
3:          Wybierz parę wyrażeń (expr_1, expr_2).
4:          jeżeli expr_1 i expr_2 są identyczne to
5:              Nie dodawaj nic do listy substytucji
6:          jeżeli expr_1 jest zmienną to
7:              jeżeli expr_1 nie występuje w expr_2 to
8:                  Dodaj do listy substytucji expr_1/expr_2
9:              w przeciwnym razie
10:                 break                              ▷ (niepowodzenie)
11:         jeżeli expr_2 jest zmienną to
12:             jeżeli expr_2 nie występuje w expr_1 to
13:                 Dodaj do listy substytucji expr_2/expr_1
14:             w przeciwnym razie
15:                 break                              ▷ (niepowodzenie)
16:         jeżeli expr_1 = f(r_1,...,r_n) i expr_2 = f(s_1,...,s_n) to ▷ oba wyrażenia są złożone i mają
    taką samą liczbę argumentów
17:             Dodaj do listy substytucji wynik Unifikuj ([r_1,...,r_n] i [s_1,...,s_n] )   ▷ składniki
    wyrażeń
18:         w przeciwnym razie   ▷ oba wyrażenia są złożone i mają inną liczbę argumentów
19:             break                                  ▷ (niepowodzenie)
20:     jeżeli wszystkie pary unifikowalne to
21:         zwróć Lista Substytucji
22:     w przeciwnym razie
23:         zwróć Niepowodzenie
```

2. Inny przykład: najbardziej ogólnym unifikatorem $f(x,y,g(y))$ i $f(z,h(z),w)$ jest $\{x/z, y/h(z), w/g(h(z))\}$

■

■ **Przykład 9.2** Przykłady niemożliwej unifikacji

1. **Wyrażenia do Unifikacji:**
   (1) Przyjaciel(Geralt, x)
   (2) Przyjaciel(Yennefer, Triss)

   Proces Unifikacji:
   - W pierwszym wyrażeniu x jest zmienną, a Geralt odnosi się do postaci z książki.
   - W drugim wyrażeniu Yennefer i Triss to konkretne postaci.
   - Próba unifikacji napotyka na problem, ponieważ Geralt i Yennefer są różnymi postaciami/stałymi.

   Wynik Unifikacji: Niepowodzenie

2. Inny przykład: Termy $f(x,g(x))$ i $f(y,y)$ nie są unifikowalne – zbiór substytucji jest pusty.

■

9.4.2 Reguły wnioskowania

Oprócz unifikacji do wnioskowania potrzebne są reguły/zasady, które w danym języku formalnym określą jak poprawnie wyprowadzać wnioski ze zbioru zdań/wyrażeń zawartych w bazie wiedzy. Reguły wnioskowania w logice umożliwiają konstrukcję poprawnych argumentów i dowodów.

W logice predykatów pierwszego rzędu istnieje kilka fundamentalnych reguł wnioskowania. Zostaną one zaprezentowane w postaci:

$$\frac{P_1, P_2, \ldots, P_n}{Q}\theta \tag{9.6}$$

gdzie $P_1, P_2, \ldots, P_n$ są wyrażeniem z przesłanki, a Q jest wnioskiem. Oznacza to, że jeśli przesłanki $P_1, P_2, \ldots, P_n$ są prawdziwe (TRUE), to z pomocą reguły można wywnioskować, że Q jest również prawdziwe (TRUE). θ jest listą substytucji uzyskanych w wyniku unifikacji.

Poniżej przedstawiono najpopularniejsze z reguł wnioskowania stosowanych w rachunku predykatów pierwszego rzędu.

– **Modus ponens**

$$\frac{P(a_1,\ldots,a_n), P(a_1,\ldots,a_n) \rightarrow Q(b_1,\ldots,b_m)}{Q(b_1,\ldots,b_m)}\theta \tag{9.7}$$

Czytaj: Jeżeli $P(a_1,\ldots,a_n)$ jest prawdziwe, i prawdą jest, że z $P(a_1,\ldots,a_n)$ wynika $Q(b_1,\ldots,b_m)$, to prawdziwe jest $Q(b_1,\ldots,b_m)$. Prawdziwość konkluzji jest warunkowana poprawnością Unifikacji dla argumentów relacji P i Q, gdzie θ to zbiór substytucji.

■ Przykład 9.3

$$\frac{\text{Wiedzmin(Geralt)}, \forall x(\text{Wiedzmin}(x) \rightarrow \text{Mutant}(x))}{\text{Mutant(Geralt)}}(x/\text{Geralt})$$

Wnioskujemy, że Geralt jest mutantem po zastosowaniu Unifikacji z listą substytucji: (x/Geralt). ■

– **Modus tollens**

$$\frac{\neg Q(b_1,\ldots,b_m), P(a_1,\ldots,a_n) \rightarrow Q(b_1,\ldots,b_m)}{\neg P(a_1,\ldots,a_n)}\theta \tag{9.8}$$

Czytaj: Jeżeli wiemy, że $\neg Q(b_1,\ldots,b_m)$ jest prawdziwe, i prawdą jest, że z $P(a_1,\ldots,a_n)$ wynika $Q(b_1,\ldots,b_m)$, to prawdziwe jest $\neg P(a_1,\ldots,a_n)$. Prawdziwość konkluzji jest warunkowana poprawnością Unifikacji dla argumentów relacji P i Q, gdzie θ to zbiór substytucji.

■ **Przykład 9.4**

$$\frac{\neg\text{Mutant(Yennefer)}, \forall x(\text{Wiedzmin}(x) \rightarrow \text{Mutant}(x))}{\neg\text{Wiedzmin(Yennefer)}}(x/\text{Yennefer})$$

Wnioskujemy, że Yennefer nie jest wiedźminem, bo wiemy, że nie jest mutantem. Wniosek poprawny po zastosowaniu Unifikacji z listą substytucji: (x/Yennefer). ■

– **Modus ponendo tollens**

$$\frac{\neg(P(a_1,\ldots,a_n) \wedge Q(b_1,\ldots,b_m)), P(a_1,\ldots,a_n)}{\neg Q(b_1,\ldots,b_m)}\theta \tag{9.9}$$

Czytaj: Jeżeli wiemy, że nie może jednocześnie być spełnione $P(a_1,\ldots,a_n)$ i $Q(b_1,\ldots,b_m)$, i $P(a_1,\ldots,a_n)$ jest prawdą, to wnioskujemy o tym, że $\neg Q(b_1,\ldots,b_m)$ jest prawdziwe. Prawdziwość konkluzji jest warunkowana poprawnością Unifikacji dla argumentów relacji P i Q, gdzie θ to zbiór substytucji.

■ **Przykład 9.5**

$$\frac{\neg(\exists x(\text{Wiedzmin(x)} \wedge \text{Czarodziej}(x))), \text{Wiedzmin}(\textit{Geralt})}{\neg\text{Czarodziej(Geralt)}}(x/\text{Geralt})$$

Wnioskujemy, że Geralt nie jest czarodziejem, bo jest wiedźminem, a reguła mówi, że nie można być jednocześnie czarodziejem i wiedźminem. Wniosek prawdziwy po zastosowaniu Unifikacji z listą substytucji: (x/Geralt). ■

– **Sylogizm dysjunkcyjny lub Modus tollendo ponens**

$$\frac{P(a_1,\ldots,a_n) \vee Q(b_1,\ldots,b_m), \neg P(a_1,\ldots,a_n)}{Q(b_1,\ldots,b_m)}\theta \tag{9.10}$$

Czytaj: Tylko jedna z alternatyw może być prawdziwa. Prawdziwość konkluzji jest warunkowana poprawnością Unifikacji dla argumentów relacji P i Q, gdzie θ to zbiór substytucji.

■ **Przykład 9.6**

$$\frac{\forall x(\text{Wiedzmin(x)} \vee \text{Czarodziej}(x)), \neg\text{Wiedzmin(Yennefer)}}{\text{Czarodziej(Yennefer)}}(x/\text{Yennefer})$$

Wnioskujemy, że Yennefer jest czarodziejką, skoro wiemy, że Yennefer nie jest wiedźminem, gdyż można być wiedźminem lub czarodziejem. Wniosek prawdziwy po zastosowaniu Unifikacji z listą substytucji: (x/Yennefer). ■

– **Reguła rezolucji** Komplementarność w logice pierwszego rzędu oznacza, że wyrażenie unifikuje się z negacją drugiego wyrażania.
Reguła rezolucji eliminuje komplementarne formuły atomowe.

$$\frac{P_1(a_1,\ldots,a_n) \vee Q_1(b_1,\ldots,b_m), \neg P_1(c_1,\ldots,c_n) \vee Q_2(d_1,\ldots,d_m)}{Q_1(b_1,\ldots,b_m) \vee Q_2(d_1,\ldots,d_m)}\theta \quad (9.11)$$

Prawdziwość konkluzji jest warunkowana poprawnością Unifikacji dla argumentów relacji P_1, Q_1 i Q_2, gdzie θ to zbiór substytucji. $P_1(a_1,\ldots,a_n)$ jest komplementarne do $\neg P_1(c_1,\ldots,c_n)$. Wyrażenia komplementarne ulegają eliminacji. Wynik rezolucji nazywa się **resolwentą**.

■ **Przykład 9.7**

$$\frac{\text{Wiedzmin(Geralt)} \vee \neg\text{Potwor}(x), \neg\text{Wiedzmin(y)} \vee \text{Mutant(y)}}{\neg\text{Potwor}(x) \vee \text{Mutant(Geralt)}}(y/\text{Geralt})$$

Eliminujemy komplementarne predykaty Wiedzmin(Geralt) i ¬Wiedzmin(y) i wnioskujemy o pozostałych predykatach z wyrażenia. Wniosek prawdziwy po zastosowaniu Unifikacji z listą substytucji: (y/Geralt). ■

9.4.3 Wnioskowanie przez łańcuchowanie progresywne i regresywne

Reguły wnioskowania stanowią podstawę do weryfikacji spełnialności lub niespełnialności wyrażeń logicznych. Aby zautomatyzować proces wnioskowania konieczna jest procedura wykonująca dowodzenie na całej bazie wiedzy. W systemach automatycznego dowodzenia i w systemach ekspertowych wyróżnia się dwa główne podejścia proceduralne do dowodzenia:

1. łańcuchowanie progresywne (ang. *forward-chaining*) nazywane też wnioskowaniem z danych/faktów;
2. łańcuchowanie regresywne (ang. *backward-chaining*) nazywane też wnioskowaniem od celu.

Łańcuchowanie progresywne

Łańcuchowanie progresywne można wyobrazić sobie jako przechodzenie po wszystkich regułach bazy wiedzy dla wszystkich znanych, zadeklarowanych faktów. Zakładając, że fakty mają status prawda (TRUE) możemy wnioskować o prawdziwości wszystkich konkluzji z reguł, których przesłanki posiadają formuły unifikowane ze tymi faktami. Proces wnioskowania w przód jest kontynuowany tak długo, aż wyczerpią się fakty i reguły do dowodzenia. Jeżeli w wyniku wnioskowania reguła

wytworzy nowy fakt, to zostaje on dopisany do bazy wiedzy i użyty do dalszego wnioskowania.

Ogólną koncepcję łańcuchowania progresywnego w logice predykatów przedstawia Algorytm 14.

Algorytm 14 Algorytm wnioskowania progresywnego [RN09]

```
1:  function LP-WP(KB, α)                          ▷ KB - baza wiedzy, α zapytanie - formuła atomowa
2:      dopóki TRUE wykonaj
3:          nowe ← {}                               ▷ Zbiór zdań utworzonych w nowej iteracji
4:          dla wszystkich r ∈ KB wykonaj
5:              (p1 ∧ ... ∧ pn → q) ← STANDARYZUJ-ZMIENNE(r)
6:              dla wszystkich θ takie, że (p1 ∧ ... ∧ pn)θ = (p'1 ∧ ... ∧ p'n)θ wykonaj   ▷ p'i z KB
7:                  q' ← PODSTAW(θ, q)                                  ▷ wnioski z listą substytucji
8:                  jeżeli q' nie unifikuje się z żadnym zdaniem z KB lub nowe to
9:                      Dodaj q' do nowe
10:                     φ ← UNIFIKUJ(q', α)
11:                     jeżeli φ ≠ fail to
12:                         zwróć φ
13:         jeżeli nowe = {} to
14:             zwróć FALSE
15:         Dodaj nowe do KB
```

Ze względu na zagnieżdżenie trzech pętli jest bardzo mało wydajny dla dużych baz wiedzy. Wewnętrzna pętla łączy każdą regułę z każdym faktem, a po uaktualnienia stanu listy faktów znów sprawdza się wszystkie reguły. Dlatego stosuje się różne zabiegi zmniejszające złożoność obliczeniową. Jedną z technik jest wybieranie tylko tych reguł, które mają w przesłance fakty dostępne z listy faktów. Ze względu na znaczący udział unifikacji w procedurze, można optymalizować proces przez szukanie tylko formuł, gdzie argumenty stałe (termy) są zgodne. Obie techniki dokładają ograniczenia (warunki), które w konsekwencji zredukują liczbę operacji. Innym podejściem jest przyrostowe łańcuchowanie progresywne. Oznacza to, że fakty, które zostały użyte w iteracji $i-1$ mogą być użyte w kolejnej iteracji tylko w połączeniu z faktami utworzonymi w iteracji i. Innymi słowy w poprzednich iteracjach wykorzystane zostały już wszystkie kombinacje i reguły dla faktów $i-1$, dlatego ich ponowne przetworzenie nie wniesie nic nowego. Tylko złożenie z nowo wytworzonymi faktami może skutkować nowymi faktami. [RN09]

■ **Przykład 9.8** Rozważmy przykład prostej bazy wiedzy, by pokazać jak działa łańcuchowanie progresywne. Rozważmy następującą historyjkę o wiedźminie:

Geralt mieszka w zamku Kaer Morhen. Mieszkańcy tego zamku uczą się skutecznej walki, by zostać wojownikami. Ponadto w pewnym momencie przechodzą oni też próbę traw. Po próbie traw, stają się wiedźminami.

Cel wnioskowania: *Chcemy dowieść, że Geralt jest wiedźminem.*

Zapiszmy historię w logice predykatów.

Fakt F1 Mieszka(Geralt, Kaer_Morhen)
Reguła R1 $\forall x(\text{Mieszka}(x, \text{Kaer\_Morhen}) \rightarrow \text{Wojownik}(x))$
Reguła R2 $\forall x(\text{Mieszka}(x, \text{Kaer\_Morhen}) \rightarrow \text{Proba}(x, \text{traw}))$
Reguła R3 $\forall x(\text{Proba}(x, \text{traw}) \rightarrow \text{Wiedzmin}(x))$
Cel G1 Wiedzmin(Geralt)

Wnioskowanie będzie przebiegać w następujących krokach:

1. Zaczynamy od znanego faktu **F1** i stosujemy go z reguła **R1**. Zastosowanie dedukcji i unifikacji z substytucją (x/Geralt) wyprowadza nowy fakt. Dodajemy nowy fakt do bazy wiedzy. Sprawdzamy czy unifikuje się z **G1** (linia 10 Algorytmu 14), co zakończyłoby wnioskowanie. Nowy fakt do pisane do KB:
   F2: Wojownik(Geralt)
2. Kontynuujemy dopasowanie **F1** do kolejnej reguły **R2**. Zgodnie z regułą modus ponens i unifikacją z substytucją (x/Geralt) uzyskujemy fakt:
   F3: Proba(Geralt, traw)
   Dodajemy nowy fakt do bazy wiedzy. Sprawdzamy czy unifikuje się z **G1** (linia 10 Algorytmu 14).
3. Próba zastosowania faktu **F1** z regułą **R3** zawodzi, gdyż przesłanka tego faktu nie pasuje do predykatu z faktu.
4. Posługujemy się teraz faktem **F2**, który nie pasuje do przesłanki żadnej z trzech reguł.
5. Używamy faktu **F3**, który nie pasuje do predykatów w przesłankach reguł **R1** i **R2**, ale pasuje do **R3**. Spełnialność przesłanki oznacza prawdziwość konkluzji, zatem powstaje nowy fakt, przy liście substytucji (x/Geralt):
   F4: Wiedzmin(Geralt). Dodajemy nowy fakt do bazy wiedzy. Sprawdzamy czy unifikuje się z **G1** (linia 10 Algorytmu 14). Unifikacja jest pozytywna zatem zwracamy wartość TRUE.

Dowiedliśmy tym samym, że Geralt jest wiedźminem. ■

Łańcuchowanie regresywne

Łańcuchowanie regresywne to alternatywny sposób wnioskowania w bazach wiedzy. Zaczynamy od celu wnioskowania (pewnej hipotezy do udowodnia) aż do uzyskania faktu, który potwierdzi dowód. Poszukuje się reguł, których konkluzje są predykatami celu wnioskowania. Z przesłanki takiej reguły powstaje nowa hipoteza do udowodnienia. Proces poszukiwania reguł lub faktów wspierających kolejne hipotezy do udowodnienia jest kontynuowany do momentu znalezienia w bazie faktu zgodnego (unifikowalnego). Przy założeniu, że baza wiedzy zawiera zbiór aksjomatów, będzie można potwierdzić wszystkie zstępne hipotezy oraz hipotezę

pierwotną.

Ogólną koncepcję wnioskowania regresywnego przedstawia Algorytm 15.

Algorytm 15 Algorytm wnioskowania regresywnego [RN09]

```
1: function LP-WR(KB, cele, θ)  ▷ KB - baza wiedzy, cele lista hipotez (θ już zastosowane), θ - lista substytucji, zwraca się pełną listę podstawień
2:     odpowiedzi ← {}
3:     jeżeli cele są puste to
4:         zwróć {θ}
5:     q' ← SUBST(θ, PIERWSZY(cele))
6:     dla wszystkich r ∈ KB ,gdzie STANDARYZUJ-ZMIENNE(r) = (p1 ∧ ... ∧ pn → q) i θ' ← UNIFIKUJ(q, q') się powiedzie wykonaj
7:         noweCele ← [p1, ..., pn | RESZTA(cele)]
8:         odpowiedzi ← LP-WR(KB, noweCele, POLACZ(θ', θ)) ∪ odpowiedzi
9:     zwróć odpowiedzi
```

Główną cechą tego algorytmu jest wykorzystanie rekurencji. Funkcja pobiera wszystkie cele, które na początku zawierają główną hipotezę do udowodnienia. Gdyby udało się spełnić wszystkie cele, to rekurencja się kończy i zwraca się listę podstawień. Algorytm operuję na pobieraniu z listy hipotez pierwszej i w pętli analizuje resztę listy hipotez. Wnioskowanie przez łańcuchowanie regresywne działa jak algorytm przeszukiwania w głąb. Jego złożoność pamięciowa jest liniowa względem wielkości dowodu. Wadą algorytmu jest powtarzanie stanów, co może prowadzić do nieskończonej rekurencji. Pomimo tego jest to główna procedura stosowane w językach programowania opartych na logice np. języku Prolog.

■ **Przykład 9.9** Rozważmy przykład wnioskowania regresywnego przy użyciu historyki i bazy wiedzy przedstawionej w przykładzie dla wnioskowania progresywnego 9.8.

Wnioskowanie będzie przebiegać w następujących krokach:

1. Zaczynamy od hipotezy **G1** i szukamy najpierw faktu, a potem reguły, która w konkluzji ma formułę zgodną z formułą celu. Sprawdzamy kolejno: **F1, R1, R2 i R3**. Pasuje reguła **R3**. Zastosowanie unifikacji z substytucją (x/Geralt) wprowadza nową hipotezę $G2$:
   G2: Proba(Geralt, traw)
2. Kontynuujemy z nowym celem **G2** i szukamy faktu lub reguły, która w konkluzji ma formułę zgodną z formułą celu. Sprawdzamy kolejno: **F1, R1, R2**. Pasuje reguła **R2**. Uzyskujemy unifikację z substytucją (x/Geralt) zyskując nową hipotezę **G3**.
   G3: Mieszka(Geralt, Kaer Mohren)

3. Kontynuujemy z nowym celem **G3** i szukamy faktu lub reguły, która w konkluzji ma formułę zgodną z formułą celu. Sprawdzamy: **F1**. W tym kroku nie generuje się nowego celu. Fakt **F1** potwierdził spełnialność celu **G3**. Co oznacza, że cel **G2** i **G1** mają status TRUE .

Dowiedliśmy tym samym, że Geralt jest wiedźminem. ■

Jak widać dowód pomija zupełnie wykorzystanie reguły **R1**, czym różni się od podejścia progresywnego. Zysk jest tym większy im większa jest liczba faktów początkowych.

9.4.4 Wnioskowanie z użyciem reguły rezolucji

Dowodzenie przez zastosowanie reguły rezolucji, prowadzi do zupełnego algorytmu wnioskowania z wykorzystaniem dowolnego zupełnego algorytmu wyszukiwania. Systemy dowodzenia twierdzeń oparte na rezolucji w rachunku predykatów pierwszego rzędu konwertuje formułę pierwszego rzędu do formy klauzulowej przed procedurą dowodzenia.

Postać klauzulowa

Użyteczność formy klauzulowej polega na tym, że upraszcza ona składnię formuł zawartych w KB. Kwantyfikatory są pomijane, a także część operatorów logicznych. Ostatecznie ma się do czynienia tylko ze zbiorami formuł atomowych. Prostota formy sprawia, że forma klauzulowa nadaje się do implementacji komputerowej systemów dowodzenia twierdzeń. Postać klauzulowa, nazywana jest koniunkcyjną postacią normalną (ang. Conjunctive normal form (CNF)).

Każde wyrażenie logiki predykatów pierwszego rzędu można przekształcić na wyrażenie w formie klauzulowej, które jest logicznie równoważne. Przekształcenie na formę klauzulową obejmuje dziewięć kroków:

1. Eliminacja równoważności zgodnie z tautologią z listy punkt 1j) ze strony 200
2. Eliminacja implikacji zgodnie z tautologią z listy punkt 1k) ze strony 200
3. Wyprowadzenie negacji zgodnie z tautologiami z listy
   (a) podwójne zaprzeczenie, punkt 1c) (str. 200)
   (b) I prawo de Morgana, punkt 1d) (str. 200)
   (c) II prawo de Morgana, punkt 1e) (str. 200)
   (d) prawo de Morgana kwantyfikator ogólny 2(a)i) (str. 200)
   (e) prawo de Morgana kwantyfikator egzystencjalny, punkt 2(a)ii) (str. 200)
4. Standaryzacja zmiennych - zmienne związane kwantyfikatorami powinny mieć różne identyfikatory dla każdego kwantyfikatora (pozwoli to uniknąć dwuznaczności przy opuszczaniu kwantyfikatora).
5. Skolemizacja: Kwantyfikatory egzystencjalne są eliminowane przez zastą-

pienie wyrażeń w formie $\forall x \exists y A(x,y)$ przez $\forall x A(x, f(x))$, gdzie x to zmienna kwantyfikowana uniwersalnie, której zakres obejmuje wyrażenie A, a f jest nowym symbolem funkcji nazywaną funkcją Skolema.
6. Przeniesienie kwantyfikatorów uniwersalnych na początek wyrażenia.
7. Usunięcie kwantyfikatorów uniwersalnych.
8. Przekształcenie wyrażenia na koniunkcję poprzez zastosowanie prawa rozdzielności dysjunkcji względem koniunkcji punkt 1m) z listy ze strony 200.
9. Wyodrębnienie sum — klauzul poprzez rozerwanie wyrażań na łącznikach $\wedge$. Klauzule zamykamy w nawiasy klamrowe.

Własność postaci klauzulowej:
Jeżeli oryginalne wyrażenie było spełnialne, to jego postać klauzulowa jest również spełnialna.

Wyjaśnimy działanie algorytmu konwersji do postaci klauzulowej na przykładzie. Rozważmy następujące zdanie: Każda czarodziejka, która zna Geralta jest w nim zakochana, albo uważa, że inne kobiety, które kochają się w jakimś wiedźminie są naiwne. Możliwa forma tego zdania w logice predykatów:

$$\begin{aligned}\forall x(\text{Czarodziej}(x) \wedge \text{Kobieta}(x) \wedge \text{Zna}(x,\text{Geralt}) \rightarrow \\ (\text{Kocha}(x,\text{Geralt}) \vee \\ (\forall y(\exists z(\text{Kobieta}(y) \wedge \text{Wiedzmin}(z) \wedge \text{Kocha}(y,z))) \rightarrow \\ \text{Ocenia}(x,y,\text{naiwna}))))\end{aligned}$$

Dokonajmy przekształcenia zdania na postać klauzulową:

1. Eliminacja równoważności: brak jest $\leftrightarrow$— krok pomijany
2. Eliminacja implikacji

$$\begin{aligned}\forall x(\neg(\text{Czarodziej}(x) \wedge \text{Kobieta}(x) \wedge \text{Zna}(x,\text{Geralt})) \vee \\ (\text{Kocha}(x,\text{Geralt}) \vee \\ (\forall y \neg(\exists z(\text{Kobieta}(y) \wedge \text{Wiedzmin}(z) \wedge \text{Kocha}(y,z))) \vee \\ \text{Ocenia}(x,y,\text{naiwna}))))\end{aligned}$$

3. Wyprowadzenie negacji zgodnie z tautologiami z listy
   (a) podwójne zaprzeczenie: brak — krok pomijamy

(b) I prawo de Morgana

$$\forall x(\neg\text{Czarodziej}(x) \vee \neg\text{Kobieta}(x) \vee \neg\text{Zna}(x,\text{Geralt}) \vee (\text{Kocha}(x,\text{Geralt}) \vee (\forall y \neg(\exists z(\text{Kobieta}(y) \wedge \text{Wiedzmin}(z) \wedge \text{Kocha}(y,z))) \vee \text{Ocenia}(x,y,\text{naiwna}))))$$

(c) II prawo de Morgana: brak zależności — krok pomijamy

(d) prawo de Morgana kwantyfikator ogólny: brak zależności — krok pomijamy

(e) prawo de Morgana kwantyfikator egzystencjalny, następnie I prawo de Morgana

$$\forall x(\neg\text{Czarodziej}(x) \vee \neg\text{Kobieta}(x) \vee \neg\text{Zna}(x,\text{Geralt}) \vee (\text{Kocha}(x,\text{Geralt}) \vee (\forall y(\forall z(\neg\text{Kobieta}(y) \vee \neg\text{Wiedzmin}(z) \vee \neg\text{Kocha}(y,z))) \vee \text{Ocenia}(x,y,\text{naiwna}))))$$

4. Standaryzacja zmiennych: każda zmienna związana kwantyfikatorem ma inny identyfikator — krok pomijamy
5. Skolemizacja: brak kwantyfikatora egzystencjalnego — krok pomijamy
6. Przeniesienie kwantyfikatorów uniwersalnych na początek wyrażenia.

$$\forall x \forall y \forall z(\neg\text{Czarodziej}(x) \vee \neg\text{Kobieta}(x) \vee \neg\text{Zna}(x,\text{Geralt}) \vee \text{Kocha}(x,\text{Geralt}) \vee \neg\text{Kobieta}(y) \vee \neg\text{Wiedzmin}(z) \vee \neg\text{Kocha}(y,z) \vee \text{Ocenia}(x,y,\text{naiwna}))$$

7. Usunięcie kwantyfikatorów uniwersalnych.

$$\neg\text{Czarodziej}(x) \vee \neg\text{Kobieta}(x) \vee \neg\text{Zna}(x,\text{Geralt}) \vee \text{Kocha}(x,\text{Geralt}) \vee \neg\text{Kobieta}(y) \vee \neg\text{Wiedzmin}(z) \vee \neg\text{Kocha}(y,z) \vee \text{Ocenia}(x,y,\text{naiwna})$$

8. Prawo rozdzielności dysjunkcji względem iloczynu: krok pomijamy
9. Wyodrębnienie dysjunkcji, klauzul: w wyniki przekształcenia powstaje jedna klauzula

$$\{\neg\text{Czarodziej}(x) \vee \neg\text{Kobieta}(x) \vee \neg\text{Zna}(x,\text{Geralt}) \vee \text{Kocha}(x,\text{Geralt}) \vee \neg\text{Kobieta}(y) \vee \neg\text{Wiedzmin}(z) \vee \neg\text{Kocha}(y,z) \vee \text{Ocenia}(x,y,\text{naiwna})\}$$

Transformacja na postać klauzulową dotyczy zawsze całej bazy wiedzy. Przekształćmy zatem na postać klauzulową wszystkie wyrażenia z tabeli 9.2 203. W prawej kolumnie tabeli 9.3 zapisano wszystkie klauzule jako wynik transformacji. W przedstawionym przykładzie uzyskano z każdej formuły logicznej dokładnie jedną klauzulę.

Tabela 9.3: Przykładowa baza wiedzy przed i po transformacji na postać klauzulową.

Lp.	Wyrażenie logiczne (KB)	Postać klauzulowa (SKB)
1	$\text{Wojownik}(\text{Geralt})$	$\{\text{Wojownik}(\text{Geralt})\}$
2	$\text{Mieszka}(\text{Geralt, Kaer Morhen})$	$\{\text{Mieszka}(\text{Geralt, Kaer Morhen})\}$
3	$\forall x(\text{Mieszka}(x, \text{Kaer Morhen}) \rightarrow \text{Wiedzmin}(x))$	$\{\neg\text{Mieszka}(x, \text{Kaer Morhen}) \vee \text{Wiedzmin}(x)\}$
4	$\text{Potwor}(\text{Bazyliszek})$	$\{\text{Potwor}(\text{Bazyliszek})\}$
5	$\forall x(\text{Wiedzmin}(x) \rightarrow \text{Mord\_Na\_Zlecenie}(x, \text{Bazyliszek}) \vee \text{Obojetny}(x, \text{Bazyliszek}))$	$\{\neg\text{Wiedzmin}(x) \vee \text{Mord\_Na\_Zlecenie}(x, \text{Bazyliszek}) \vee \text{Obojetny}(x, \text{Bazyliszek})\}$
6	$\forall x(\exists y(\text{Obojetny}(x, y)))$	$\{\text{Obojetny}(x, S(x))\}$ *(S(x) funkcja Skolema)*
7	$\forall x \forall y(\text{Wojownik}(x) \wedge \text{Potwor}(y) \wedge \text{Walka}(x, y) \rightarrow \neg\text{Obojetny}(x, y))$	$\{\neg\text{Wojownik}(x) \vee \neg\text{Potwor}(y) \vee \neg\text{Walka}(x, y) \vee \neg\text{Obojetny}(x, y)\}$
8	$\text{Walka}(\text{Geralt}, \text{Bazyliszek})$	$\{\text{Walka}(\text{Geralt}, \text{Bazyliszek})\}$

Algorytm rezolucji

Przed omówieniem algorytmu rezolucji rozważmy pojęcia związane z automatycznym dowodzeniem twierdzeń.

Definicja 9.4.1 Interpretacje Herbranda

Interpretacja (lub struktura) dowolnego predykatu lub funkcji oznaczonego symbolem σ składa się z:

- niepustego zbioru $|I|$, nazywanego uniwersum (lub dziedziną) I,
- dla każdej stałej c predykatu/funkcji σ, element c^I jest elementem zbioru $|I|$,
- dla każdej stałej funkcji f predykatu/funkcji σ o liczbie argumentów $n > 0$, funkcja f^I to mapowanie z $|I|^n$ do $|I|$,
- dla każdej stałej P predykatu/funkcji σ, element P^I zbioru ma wartość $\{\text{FALSE}, \text{TRUE}\}$,
- dla każdej stałej R predykatu/funkcji σ o liczbie argumentów $n > 0$, funkcja R^I mapuje z $|I|^n$ do $\{\text{FALSE}, \text{TRUE}\}$.

Interpretacje Herbranda jest istotna w automatycznym dowodzeniu twierdzeń. Są one definiowane dla zbioru klauzul S. Dziedzina D interpretacji Herbranda I składa się ze zbioru termów zawierających symbole funkcji i stałe. Symbole stałe i funkcji są interpretowane tak, że dla każdego skończonego termu t utworzonego z tych symboli, t^I jest samym termem t, który jest elementem D. Załóżmy że S zawiera jednowymiarową funkcję oznaczoną symbolem F i stałą oznaczoną C. Wówcza $D = \{C, F(C), F(F(C)), F(F(F(C))), \ldots\}$ i C jest interpretowane tak, że C^I jest elementem C z D, a F jest interpretowane tak, że F^I dla termu C daje term $F(C)$, F^I stosowane do termu $F(C)$ z D daje $F(F(C))$, i tak dalej. Nie ma ograniczeń co do tego, w jaki sposób interpretacja Herbranda I może interpretować symbole predykatów S. [HLP08]

Zainteresowanie interpretacjami Herbranda dla dowodzenia twierdzeń wynika z następującego twierdzenia:

Twierdzenie 9.4.1 Jeśli S jest zbiorem klauzul, to S jest spełnialne wtedy i tylko wtedy, gdy istnieje interpretacja Herbranda I taka, że $I \models S$ (I jest modelem S).

Innymi słowy jeżeli $S^I = TRUE$

Twierdzenie to oznacza, że do celów testowania spełnialności zbiorów klauzul wystarczy rozważyć interpretacje Herbranda. Prowadzi to pośrednio do automatycznej procedury dowodzenia twierdzeń, która jest oparta na weryfikacji logicznych konsekwencji, przez wielokrotne obliczenia, dla każdej formuły i stałej. Ze względu na wykładniczą złożoność czasową rozważa się inne podejście do automatycznego dowodzenia oparte na niespłnialności.

Metoda dowodzenia twierdzeń jest uznawana za zupełną, jeśli jest w stanie udowodnić każdą ważną formułę. W przypadku testowania niespełnialności, metoda dowodzenia twierdzeń jest uznawana za zupełna, jeśli może wyprowadzić fałsz, czyli klauzulę pustą, z każdego niespełnialnego zbioru klauzul. Wiadomo, że rezolucja ma taką własność.

Twierdzenie 9.4.2 Zbiór S klauzul pierwszego rzędu jest niespełnialny wtedy i tylko wtedy, gdy istnieje sprzeczność (klauzula pusta $\{\}$) wynikająca z rezolucji z S.

Załóżmy, że istnieje ogólny zbiór A aksjomatów oraz szczególna formuła F, którą chce się udowodnić. Dowód polega na wykazaniu, że formuła $A \rightarrow F$ jest prawdziwa/spełnialna. W podejściu dowodzenia przez zaprzeczenie, wykazuje się, że $\neg(A \rightarrow F)$ jest niespełnialna. Przez przekształcenie $\neg(A \rightarrow F)$ na formę klauzulową uzyskujemy: $A \wedge \neg F$, Ze zbioru aksjomatów A uzyskuje się zbiór klauzul S i z formuły $\neg F$ uzyskuje się oraz zbiór klauzul D. Zbiór$F \cap SF$ jest

niespełnialny wtedy i tylko wtedy, gdy $A \rightarrow F$ jest ważne/prawdziwe.

Załóżmy, że dowodzimy formułę logiczną D ze zbioru wyrażeń stanowiących bazę wiedzy KB (innymi słowy sprawdzamy spełnialność D w KB). Przed zastosowaniem algorytmu rezolucji konieczne jest wykonanie następujących kroków:

1. transformacja bazy wiedzy KB na postać klauzulową normalną ze zbiorem klauzul S
2. transformacja $\neg D$ na postać normalną (klauzulę) A

Następnie stosuje się procedurę przedstawioną w postaci pseudokodu Algorytm 16.

Algorytm 16 Algorytm dowodzenia rezolucji

```
1:  procedura REZOLUCJA(S)                                   ▷ S jest zbiorem klauzul
2:      dopóki nie znaleziono klauzuli pustej i S nie jest pusty wykonaj
3:          Wybierz pary klauzul z S
4:          dla każdej pary klauzul (C1,C2) wykonaj
5:              C_resolwenta ← ZastosujRezolucję(C1,C2)
6:              jeżeli C_resolwenta jest klauzulą pustą to
7:                  zwróć TRUE
8:              jeżeli C_resolwenta ⊆ S to
9:                  zwróć FALSE
10:             Dodaj C_resolwenta do S
11:     zwróć FALSE
1:  function ZASTOSUJREZOLUCJĘ(C1,C2)
2:      Resolventa ← FALSE
3:      C_nowa ← pusta klauzula
4:      dla każdego literału L w C1 wykonaj
5:          dla każdego literału M w C2 wykonaj
6:              U ← Unifikuj(L,M)                            ▷ Próba unifikacji literałów
7:              jeżeli U nie jest puste to
8:                  L' ← ZastosujSubstytucję(L,U)
9:                  M' ← ZastosujSubstytucję(M,U)
10:                 jeżeli L' i M' są komplementarne to
11:                     Resolventa ← TRUE
12:                     C_resolwenta ← C_resolwenta ∪ ZastosujSubstytucję(C1 \ {L} ∪ C2 \ {M},U)
13:     jeżeli Resolventa to
14:         C_resolwenta ← Faktoryzuj(C_resolwenta)          ▷ Usuń duplikaty literałów komplementarnych
15:         zwróć C_resolwenta
16:     w przeciwnym razie
17:         zwróć NULL          ▷ Nie znaleziono par unifikowalnych i komplementarnych literałów
```

Poniższy przykład prezentuje sposób działania algorytmy Rezolucji na prostym wyrażeniu. Przykład pochodzi z [HLP08].

■ **Przykład 9.10** Dana jest ważna (TRUE) formuła logiki pierwszego rzędu:
$\forall x \exists y(P(x) \rightarrow Q(x,y)) \wedge \forall x \forall y \exists z(Q(x,y) \rightarrow R(x,z)) \rightarrow \forall x \exists z(P(x) \rightarrow R(x,z))$

Dowodząc przez sprzeczność, neguje się formułę, co prowadzi do następującej formy:

$$\neg[\forall x \exists y(P(x) \to Q(x,y)) \wedge \forall x \forall y \exists z(Q(x,y) \to R(x,z)) \to \forall x \exists z(P(x) \to R(x,z))],$$

W następnym kroku wykazać, że formuła jest niespełniana. Ten krok wymaga transformacji do formy klauzulowej, co prowadzi do powstania zbioru klauzul S:

$$S = \{\{\neg P(x) \vee Q(x, f(x))\}, \{\neg Q(x,y) \vee R(x, g(x,y))\}, \{P(a)\}, \{\neg R(a,z)\}\}.$$

Dowód z użyciem rezolucji ze zbioru klauzul S:

K	Klauzula	Komentarz
1	$\{P(a)\}$	wejście
2	$\{\neg P(x) \vee Q(x, f(x))\}$	wejście
3	$\{Q(a, f(a))\}$	substytucja x/a i rezolucja dla 1, 2
4	$\{\neg Q(x,y) \vee R(x, g(x,y))\}$	wejście
5	$\{R(a, g(a, f(a)))\}$	substytucja x/a, rezolucja dla 3 i 4
6	$\{\neg R(a,z)\}$	wejście
7	$\{\}$	rezolucja 5 i 6 - sprzeczność - FALSE

W kolumnie „komentarz” wzmianka „wejście” oznacza, że klauzula znajduje się w S. Algorytm rezolucji wyprowadził fałsz (klauzulę pustą), wynika z tego, że S jest niespełnialne, co w konsekwencji oznacza, że oryginalna formuła jest spełnialna.

■

■ **Przykład 9.11** Wróćmy do przykładu z książek o wiedźminie z tabeli 9.3 ze strony 215. Użyjemy zapisanych tam klauzul do udowodnienia, że Geralt zamordował na zlecenie Bazyliszka. Do bazy klauzul dodamy zanegowaną hipotezę do udowodnienia czyli:

$$\{\neg \text{Mord\_Na\_Zlecenie(Geralt, Bazyliszek)}\}.$$

Kroki rezolucji są kontrolowane poprzez dobieranie takich klauzul, które zawierają komplementarne po unifikacji formuły atomowej. **K** w tabeli to krok algorytmu, **Klauzula** pochodzi z 9.3.

K	Klauzula	Komentarz
1	$\{\neg \text{Mord\_Na\_Zlecenie}(\text{Geralt}, \text{Bazyliszek})\}$	hipoteza
2	$\{\neg \text{Wiedzmin}(x) \vee \text{Mord\_Na\_Zlecenie}(x, \text{Bazyliszek}) \vee \text{Obojetny}(x, \text{Bazyliszek})\}$	klauzula 5 z tab. 9.3
3	$\{\neg \text{Wiedzmin}(\text{Geralt}) \vee \text{Obojetny}(\text{Geralt}, \text{Bazyliszek})\}$	substytucja x/Geralt i resolwenta z rezolucji K1, K2
4	$\{\neg \text{Mieszka}(x, \text{Kaer Morhen}) \vee \text{Wiedzmin}(x)\}$	klauzula 3 z tab. 9.3
5	$\{\neg \text{Mieszka}(\text{Geralt}, \text{Kaer Morhen}) \vee \text{Obojetny}(\text{Geralt}, \text{Bazyliszek})\}$	substytucja x/Geralt i resolwenta z rezolucji dla K3 i K4
6	$\{\text{Mieszka}(\text{Geralt}, \text{Kaer Morhen})\}$	klauzula 2 z tab. 9.3
7	$\{\text{Obojetny}(\text{Geralt}, \text{Bazyliszek})\}$	resolwenta z rezolucji dla K5 i K6
8	$\{\neg \text{Wojownik}(x) \vee \neg \text{Potwor}(y) \vee \neg \text{Walka}(x, y) \vee \neg \text{Obojetny}(x, y)\}$	klauzula 7 z tab. 9.3
9	$\{\neg \text{Wojownik}(\text{Geralt}) \vee \neg \text{Potwor}(\text{Bazyliszek}) \vee \neg \text{Walka}(\text{Geralt}, \text{Bazyliszek})\}$	substytucja x/Geralt i y/Bayliszek i resolwenta z rezolucji dla K7 i K8
10	$\{\text{Wojownik}(\text{Geralt})\}$	klauzula 1 z tab. 9.3
11	$\{\neg \text{Potwor}(\text{Bazyliszek}) \vee \neg \text{Walka}(\text{Geralt}, \text{Bazyliszek})\}$	resolwenta z rezolucji dla K9 i K10
12	$\{\text{Potwor}(\text{Bazyliszek})\}$	klauzula 4 z tab. 9.3
13	$\{\neg \text{Walka}(\text{Geralt}, \text{Bazyliszek})\}$	resolwenta z rezolucji dla 11 i 12
14	$\{\text{Walka}(\text{Geralt}, \text{Bazyliszek})\}$	klauzula 8 z tab. 9.3
15	$\{\}$	rezolucja K13 i K14 w wyniku uzyskijemy sprzeczność - FALSE

Tym samym zostało udowodnione, że Geralt zabił bazyliszka na zlecenie. ■

Wnioskowanie poprzez stosowanie algorytmu rezolucji doprowadza ostatecznie do znalezienia dowodu, jeżeli on istnieje. Efektywność algorytmu zależy od zastosowania odpowiedniej strategii. Doświadczenie zdobyte podczas pracy z dowodzeniem może przyczynić się do lepszego zrozumienia, które techniki zmniejszają złożoność obliczeniową. Niestety nie ma rozwiązań globalnie dobrych. Do znanych optymalizacji należą: [HLP08; RN09]:

– Klauzule jednostkowe (ang. unit preference) w pierwszej kolejności wybie-

rane są zdania proste składające się z pojedynczych termów. Sprawia to, że resolwenty są krótsze. Technika wymaga zastosowania tylko pojedynczego wyrażenia, co sprawia, że algorytm nie jest zupełny. Natomiast jest zupełny w zbiorze klauzul Horna, o których będzie mowa w rozdziale 10.

– Dedukcja P1 opiera się na założeniu, że rezolucja pozytywna (wykorzystująca co najmniej jedną klauzulę pozytywnej, czyli taką która nie zawiera zanegowanych termów) jest zupełna, to znaczy, że jeśli zbiór S jest niespełnialny, to istnieje sprzeczność z S, w której wszystkie rezolucje są pozytywne.
– Hiperrezolucja jest modyfikacją rezolucji pozytywnej, w której seria pozytywnych resolwent jest wykonywana naraz. Załóżmy, że C jest klauzulą mającą co najmniej jeden literał negatywny, a $D_1, D_2, \ldots, D_n$ są klauzulami pozytywnymi. Ponadto C_1 jest resolwentą z C i D_1, C_2 jest resolwentą z C_1 i D_2, itd., a C_n jest resolwentą z C_{n-1} i D_n. Zakładamy, że C_n jest klauzulą pozytywną, ale żadna z klauzul C_i nie jest pozytywna, dla $i < n$. Wtedy C_n nazywa się hiperresolwentą C i $D_1, D_2, \ldots, D_n$. Zatem w hiperrezolucji przeprowadza się sekwencje rezolucji pozytywnych. Hiperrezolucja jest czasami użyteczna, ponieważ redukuje liczbę pośrednich wyników, które muszą być przechowywane w weryfikatorze.
– Rezolucja na wejściu: Zazwyczaj próbuje się wykazać niespełnialność, gdy do zbioru klauzul aksjomatów A dodamy zanegowane wyrażenie do udowodnienia F i wówczas uruchamiamy rezolucję. Z racji tego, że dowodzi się F, można oczekiwać, że rezolucje obejmujące klauzule z F będą bardziej przydatne, ponieważ rezolucje obejmujące dwie klauzule z A są połączeniem ogólnych aksjomatów.
– Zbiory wsparcia (ang. set of support) - Jeżeli chciałoby się wykonywać tylko rezolucje obejmujące klauzule z F lub klauzule z nich pochodzące. Można to osiągnąć za pomocą strategii zbioru wsparcia, jeśli zbiór z F jest odpowiednio dobrany.

10. Język programowania Prolog

Prolog (ang. *programming in logic*) jest najpopularniejszym językiem do programowania w logice. Efektywność języka Prolog wynika z połączenia ograniczonej formy logiki pierwszego rzędu oraz specyficznych strategii rezolucji i wyszukiwania.

10.1 Połączenie składni Prologu z logiką predykatów

Przedstawimy składnię języka Prolog wychodząc od postaci klauzulowej w logice predykatów pierwszego rzędu. Załóżmy, że dana jest klauzula, w której P_i to formuła atomowa pozytywna, gdzie $1 \leq i \leq n$, a N_j to formuła atomowa zanegowane, gdzie $1 \leq j \leq m$. x symbolicznie przedstawia argumenty relacji i należy symbol utożsamić z listą termów o wymiarowości $d \geq 0$. Dokonamy ich porządkowania i klauzule pozytywne wyprzedzą klauzule negatywne:

$$(P_1(x) \vee P_2(x) \vee \cdots \vee P_n(x)) \vee (\neg N_1(x) \vee \neg N_2(x) \vee \cdots \vee \neg N_m(x))$$

Po zastosowaniu I prawa de Morgana uzyskamy następującą postać zdania:

$$(P_1(x) \vee P_2(x) \vee \cdots \vee P_n(x)) \vee (\neg(N_1(x) \wedge N_2(x) \wedge \cdots \wedge N_m(x))$$

Przechodząc z formy sumy na implikację otrzymujemy:

$$(P_1(x) \vee P_2(x) \vee \cdots \vee P_n(x)) \leftarrow (N_1(x) \wedge N_2(x) \wedge \cdots \wedge N_m(x))$$

Wprowadzamy następujące oznaczenia:
- $\wedge$ zamienimy na przecinek ,
- $\vee$ zamienimy na średnik ;
- $\leftarrow$ zamienimy na dwukropek-myślnik : $-$

i otrzymamy klauzulę postaci:

$$(P_1(x); P_2(x); \ldots; P_n(x)) : -(N_1(x), N_2(x), \ldots, N_m(x)).$$

Nazwijmy tą notację semiprologową.

Wprowadźmy nowe pojęcie reprezentujące specjalną klasę klauzul zwanych klauzulami Horna.

Definicja 10.1.1 Klauzula Horna to klauzula, która ma co najwyżej jeden pozytywny predykat. Dzieli się je na:
1. Klauzule z głową posiadające jeden pozytywny predykat.
2. Klauzule bez głowy bez pozytywnego predykatu.

Stosując nową notację, semiprologową, klauzula Horna, która może mieć tylko jedno wyrażenie pozytywne P_i, co zapisujemy jako:

$$(P(x) : -(N_1(x), N_2(x), \ldots, N_m(x)).$$

Konsekwencją tego ograniczenia jest to, że może nie być możliwe wyrażenie wiedzy kluczowej dla danej aplikacji. Implikacja, której konkluzja jest dysjunkcją, nie jest wyrażalna w formie klauzul Horna. Oznacza to, na przykład, że nie można reprezentować reguły takiej jak „Jeśli zdiagnozowano u Ciebie wysokie ciśnienie krwi, musisz albo zmniejszyć spożycie soli, albo przyjmować leki”, ponieważ najbardziej naturalnie reprezentuje się ją jako implikację z dysjunkcją w konsekwencji. Odnosząc się do przykładu o Wiedźminie z tabeli 9.3 klauzula nr 5 nie jest klauzulą Horna.

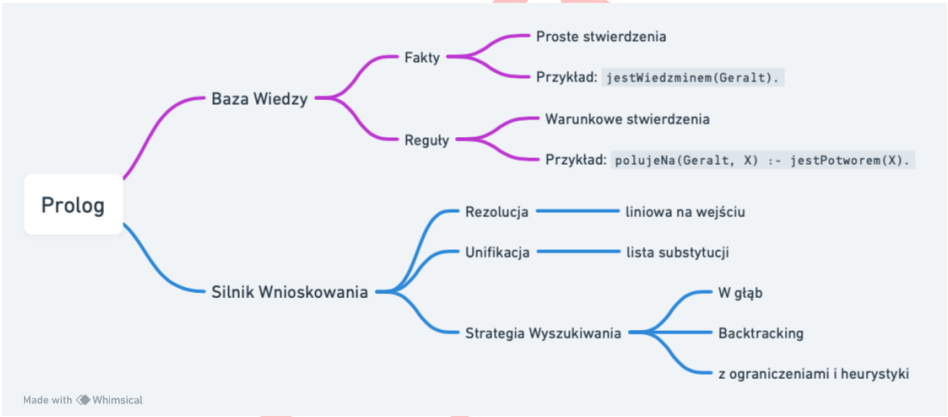
Klauzule Horna znalazły mimo to ogromne zastosowanie w językach programowania logicznego. Wynika to z własności tychże, że jeśli S jest zbiorem klauzul Horna, to rezolucja jednostkowa jest zupełna.

Wracając do języka Prolog, to prosty program w tym języku składa się ze zbioru klauzul Horna (zbioru implikacji) w notacji semiprologowej. Program opisuje rozwiązywany problem konstruując:
- **fakty**: klauzule Horna z samą głową bez treści, czyli implikacje bezwarunkowe oraz

– **reguły**: klauzule Horna z głową i z treścią, czyli implikacje warunkowe.

Zgodnie z podejściem deklaratywnym, które w przeciwieństwie do podejścia imperatywnego (opisującego kroki potrzebne do osiągnięcia określonego wyniku), kod Prologu opisuje logikę obliczeń bez ich bezpośredniej implementacji. Stosowana w Prologu strategia rezolucji to liniowa rezolucja wejściowa, a strategia wyszukiwania to łańcuchowanie regresywne, często realizowane przez algorytm przeszukiwania strategią w głąb (depth-first search).

Języki programowania w logice w tym Prolog 10.1 rozdzielone są na dwa funkcjonalne bloki: Baza Wiedzy, która jest właściwym kodem źródłowym i Silnik Wnioskowania, który jest już dany i z perspektywy programisty musi być znany, ale nie jest implementowany.



Rys. 10.1: Język programowania Prolog — dwa komponenty (przykłady podane są we właściwej składni języka Prolog a nie logiki predykatów).

10.2 Elementy składni

Poniższe podrozdziały w sposób syntetyczny wprowadzają elementy składni Prologu przedstawiając typy danych i używane operatory, sposoby tworzenia faktów i reguł, oraz przejście do trybu wnioskowania.

10.2.1 Typy danych i operatory

Prolog nie wymaga deklarowania typów. Typ zostaje rozpoznany na podstawie składni. W programowaniu można wykorzystać następujące typy danych:

– atomy: stałe wartości, identyfikator składa się z symboli i rozpoczyna się z małej litery lub jest zamknięty w apostrofy, np.: `geralt`, `x25`, `x_`, `x_y`, `'Yennefer'`,

– liczby, np.: `1`, `-97`, `3.14`, `-0.000035`;
– zmienne: może przez unifikację stać się dowolnym atomem, zmienną lub strukturą, identyfikator zmiennej rozpoczyna się z wielkiej litery lub podkreślenia, np.: `X`, `Wynik`, `_x23`; wyróżnia się specjalną zmienną anonimową: `_`;
– struktury (predykaty), które opisują relację pomiędzy argumentami; predykat ma wymiarowość $d \geq 0$, ponadto wymiarowość jest stała w obrębie danego programu, np.: `data(1, maj, 2004)`, `kobieta('Yennefer')`, `odcinek(punkt(X,Y), punkt(X1,Y2))`, `par(r1,seq(r2,r3))`;
– listy, które identyfikuje się po nawiasach kwadratowych i charakteryzuje zmienna długość oraz nie muszą być homogeniczne: `[]`, `[a, X, par(X,Y)]`.

Tabela 10.1: Lista operatorów arytmetycznych i logicznych w języku Prolog.

Operatory arytmetyczne		Operatory logiczne	
Symbol	Opis	Symbol	Opis
+	dodawanie	=:=	czy wartości liczbowe lub wyrażenia arytmetyczne są równe
−	odejmowanie	=\=	czy wartości liczbowe lub wyrażenia arytmetyczne są różne
/	dzielenie	>	większe
//	dzielenie całkowite	<	mniejsze
*	mnożenie	>=	większe równe
**	potęga	=<	mniejsze równe
mod	reszta z dzielenia		
is	znak równości (wynik obliczeń arytmetycznych)		

10.2.2 Definiowanie faktów

Fakty w języku Prolog:
– są stwierdzeniami, które są zawsze prawdziwe (TRUE) w kontekście bazy wiedzy (założenie o zamkniętym świecie),
– konkretne informacje lub asercje są zapisywane jako pojedyncze klauzule Horna bez treści (implikacja bez warunku).
– zwyczajowo nie zawierają zmiennych (lub zawierają zmienne, które są traktowane jako uniwersalnie kwantyfikowane).

Rozpatrzmy przykład deklarowania faktów w języku Prolog. Dla porównania zostaną podane interpretacje faktów w logice predykatów pierwszego rzędu, a następnie w składni języka (tabela 10.2). Predykaty `kobieta` i `mezczyzna` mają jeden argument, a predykat `lubi` ma dwa argumenty, przy czym przyjmujemy, że pierwszy to podmiot relacji, a drugi argument, to przedmiot relacji.

Tabela 10.2: Przykłady faktów w języku Prolog.

Zdanie	**Logika predykatów pierwszego rzędu**	**Język Prolog**
Yennefer to kobieta.	Kobieta(Yennefer)	`kobieta(yennefer).`
Triss Merigold to kobieta.	Kobieta(TrissMerigold)	`kobieta(triss_merigold).`
Geralt to mężczyzna.	Mezczyzna(Geralt)	`mezczyzna(geralt).`
Jaskier to mężczyzna.	Mezczyzna(Jaskier)	`mezczyzna(jaskier).`
Geralt lubi używać ironii.	Lubi(Geralt, Uzywac(Ironia))	`lubi(geralt, uzywac(ironia)).`
Yennefer lubi nosić czerń.	Lubi(Yennefer, Nosic(Czern))	`lubi(yennefer, nosic(czern)).`
Jaskier lubi pisać wiersze.	Lubi(Jaskier, Pisac(Wiersze))	`lubi(jaskier, pisac(wiersze)).`

10.2.3 Definiowanie reguł

Reguły w języku Prolog:

- służą do wyrażania zależności lub implikacji między różnymi faktami;
- są relacjami zawsze prawdziwymi w bazie wiedzy (aksjomatami);
- są klauzulami Horna, które składają się z głowy (ang. head) i treści (ang. body), gdzie treść reguły określa warunek, który musi być spełniony, aby głowa była prawdziwa;
- mogą zawierać zmienne, które są wiązane (unifikowane) podczas procesu wnioskowania.

Rozwińmy bazę wiedzy z tabeli 10.2 dodając trzy reguły. Będą one definiowały na nowo relację `lubi` dla Gerlata, Yennefer i Jaskiera. Warto zauważyć, że każda z reguł posługuje się tą samą zmienną X, co w niczym nie przeszkadza, bo zasięg zmiennej to jedna reguła lub zapytanie. Zmienne nie pamiętają swojej wartości po wykonaniu wnioskowania.

Tabela 10.3: Przykłady reguł w języku Prolog.

Zdanie	**Logika predykatów pierwszego rzędu**	**Język Prolog**
Geralt lubi kobiety.	$\forall x$(Kobieta(x) $\rightarrow$ Lubi(Geralt, x))	`lubi(geralt, X) :- kobieta(X).`
Yennefer lubi to samo co Geralt.	$\forall x$(Lubi(Yennefer, x) $\rightarrow$ Lubi(Geralt, x))	`lubi(yennefer, X) :- lubi(geralt, X).`
Jaskier lubi to samo co Yennefer.	$\forall x$(Lubi(Jaskier, x) $\rightarrow$ Lubi(Yennefer, x))	`lubi(jaskier, X) :- lubi(yennefer, X).`

10.2.4 Zadawanie zapytań — wnioskowanie

Wnioskowanie w Prologu rozpoczyna się od postawienia hipotezy, czyli jest to wnioskowanie regresywne. Hipoteza to jedyna wprowadzona do bazy wiedzy formuła bez głowy. Jest ona wystarczająca, by rozpocząć proces rezolucji.

Załóżmy, że posługujemy się bazą wiedzy w Prologu z wykorzystaniem notacji semiprologowej:

```
kobieta(yennefer):-
kobieta(triss_merigold):-
mezczyzna(geralt):-
mezczyzna(jaskier):-

lubi(geralt, uzywac(ironia)):-
lubi(yennefer, nosic(czern)):-
lubi(jaskier, pisac(wiersze)):-

lubi(geralt, X) :- kobieta(X)
lubi(yennefer, X) :- lubi(geralt, X)
lubi(jaskier, X) :- lubi(yennefer, X)
```

Przedstawmy proces wnioskowania, gdy pytamy o prawdziwość celu (ang. goal) wnioskowania/hipotezy, że Yennefer coś lubi. Zapytanie w języku Prolog będzie miało formę `lubi(yennefer, X).` przypominającą fakt.

Rozważmy własność wnioskowania przez rezolucję. Formalnie hipoteza musi zostać zanegowana. Zatem stosując notację semiprologową, zapytanie będzie miało formę:

```
:-lubi(yennefer, X),
```

gdyż nie zawiera niezanegowanego predykatu. Posiadając tylko jedną klauzulę Horna bez głowy wyzwoli się wnioskowanie z użyciem rezolucji. Algorytm zastosuje zasadę rezolucji w kolejności wpisania faktów i reguł (łańcuchowaniem regresywnym) wychodząc od zapytania/hipotezy `:-lubi(yennefer, X)`:

1. Sklejane są głowy i treści klauzul Horna o komplementarnych predykatach:

$$\frac{:-lubi(\text{yennefer},X), lubi(\text{yennefer}, nosic(czern)) :-}{:-}[X/nosic(czern)]$$

   Wynikiem rezolucji jest pusta klauzula (bez głowy i treści), przy substytucji `nosic(czern)` za X, bo taka unifikacja sprawia, że obie formuły są komplementarne. Co oznacza, potwierdzenie hipotezy i udowodnienie, że jest prawdziwa. Wynikiem jest lista substytucji X/nosic(czern).
2. Rezolucja może być kontynuowana. W kolejnym kroku użyta jest reguła, której głowa jest komplementarna z hipotezą, czyli:

$$\frac{:-lubi(\text{yennefer},X), lubi(\text{yennefer},X) :- lubi(\text{geralt},X)}{:-lubi(\text{geralt},X)}$$

   Reguła jest warunkowana prawdziwością prawej strony. W tym momencie powstaje cel cząstkowy (ang. subgoal) wnioskowania i jest on resolwentą. Staje się zapytaniem o ustalenie co lubi Geralt. Uruchamiana jest procedura rezolucji dla celu cząstkowego.
   (a) Cel cząstkowy jest komplementarny do faktu, co zapisujemy:

$$\frac{:-lubi(\text{geralt},X), lubi(\text{geralt},\text{uzywac(ironia)}) :-}{:-}[X/uzywac(ironia)]$$

   Wynikiem rezolucji jest pusta klauzula przy substytucji `X/uzywac(ironia)`, bo taka unifikacja sprawia, że struktury są komplementarne. Tym samym po raz drugi uzyskuje się potwierdzenie prawdziwości hipotezy. Wynikiem jest substutucja.
   (b) Rezolucja jest kontynuowana, z zastosowaniem reguły, której głowa jest komplementarna z celem cząstkowym wnioskowania:

$$\frac{:-lubi(\text{geralt},X), lubi(\text{geralt},X) :- kobieta(X)}{:-kobieta(X)}$$

   Reguła jest warunkowana prawdziwością prawej strony. W tym momencie powstaje kolejny cel cząstkowy wnioskowania, pytający kto jest kobietą. Uruchamiana jest procedura rezolucji dla kolejnego celu cząstkowego.

i. Rezolucja przebiega z faktem – Yennfer jest kobietą:

$$\frac{:-kobieta(X), kobieta(\text{yennefer}) :-}{:-}[X/\text{yennefer}]$$

Wynikiem rezolucji jest klauzula pusta: przy liście substytucji `X2/yennefer`, bo taka unifikacja sprawia, że struktury są komplementarne. Otrzymano kolejny zestaw podstawień, z którym hipoteza jest prawdziwa.

ii. Na podobnej zasadzie znajdowane jest rozwiązanie:

$$\frac{:-kobieta(X), kobieta(\text{triss\_merigold} :-)}{:-}[X/\text{triss\_merigold}]$$

W przedstawionym przykładzie rezolucja przynosi cztery rozwiązania w postaci listy substytucji, które zapewniają, że zapytanie wprowadzone w formie zanegowanej do bazy wiedzy składającej się z aksjomatów doprowadziły do klauzuli pustej (sprzeczności):

(1) `X = nosic(czern),`
(2) `X = uzywac(ironia),`
(3) `X = yennefer,`
(4) `X = triss_merigold.`

10.3 Cechy Prologu

Prolog wykonuje wnioskowanie z użyciem łańcuchowania regresywnego realizowanego algorytmem w głąb sprawdzając klauzule w kolejności ich pojawiania się w bazie wiedzy.

Główne cechy języka Prolog to [HLP08; RN09]:

– Bardziej restrykcyjne traktowanie założenia o unikalności nazw i założenia o zamkniętym świecie, w konsekwencji tylko zdania zgromadzone w bazie wiedzy są TRUE. Przez co nie jest możliwe udowodnienie, że zdanie jest fałszywe. Z tego też powodu symbol = równości termów jest bardziej restrykcyjny. W Prologu jest oznaczeniem sprawdzenia możliwości unifikacji w kontekście faktów i stałych podanych w bazie wiedzy lub zapytaniu. Brak równości termów z logiki powoduje, że Prolog nie może być używany do matematycznego dowodzenia twierdzeń.
– Problem z negacją. Logiczna negacja predykatów jest zaszyta w klauzulach Horna prezentujących pojedynczą regułę (początek rozdziału 10). Aby umożliwić użycie negacji zastosowano pewien wybieg. Pomaga w tym założenie o zamkniętym świecie, z pełnym opisem uniwersum. Natomiast w

sytuacjach, gdzie baza wiedzy jest niekompletna lub otwarta, podejście to może prowadzić do błędnych wniosków.
Procedura ta nazywa się negacją przez niepowodzenie (ang. Negation as Failure). Działa ona tak, że wyrażenie `not P` jest prawdziwe, jeśli Prolog nie może udowodnić `P`. Jest to podejście proceduralne, ale ma swoje ograniczenia, ponieważ opiera się na niepowodzeniu dowodu, a nie na dowodzie nieprawdziwości.

Na przykład, dana jest w Prologu reguła z wykorzystaniem wbudowanego predykatu `not`:

```
wiedzmin(X) :- not czarodziej(X).
```

Oznacza, że jeżeli `czarodziej(geralt)` nie będzie mogło być dowiedzione w bazie wiedzy, to `wiedzmin(geralt)` uzyska status TRUE.

W logice klasycznej, nieudowodnienie twierdzenia nie jest równoznaczne z dowodem jego fałszywości. Negacja przez niepowodzenie może prowadzić do błędnych wniosków, szczególnie w sytuacjach, gdzie brak wiedzy (brak dowodu) nie jest równoznaczny z fałszem. Rodzi to problemy ze zmiennymi, gdyż nie jest możliwe negowanie zapytania zawierającego zmienne, które nie zostały zunifikowane. Na przykład, zapytanie `not wiedzmin(X)` może nie działać zgodnie z oczekiwaniami, jeśli `X` nie jest zunifikowane ze stałą.

– Algorytm w głąb nie jest zupełny, co oznacza, że w pewnych sytuacjach może nie znaleźć rozwiązania, nawet jeśli takie rozwiązanie istnieje. Rekursywna implementacja algorytmu w głąb jako silnika wnioskowania w Prologu sprawia, że algorytm schodzi w dół drzewa reguł, eksplorując jak najgłębiej każdą ścieżkę, zanim przejdzie do następnej. Jeśli drzewo zawiera nieskończone ścieżki (np. w wyniku źle napisanej reguły rekurencyjnej), algorytm może utknąć na jednej ze ścieżek i nigdy nie osiągnąć rozwiązania.

Przykład rekurencji skończonej

Rozważmy zależność, wskazującą, że istnieje relacja pomiędzy bohaterami z sagi o Wiedźminie. Zagadnienie to można utożsamić z klasycznym problemem sprawdzenia, czy istnienia ścieżka w grafie pomiędzy dwoma węzłami. Zależność tą możemy zdefiniować w Prologu następująco"

(Reguła 1:) `znajomy( X, Y) :- zna( X, Y).`
`(Reguła 2:) znajomy( X, Z) :- znajomy( X, Y), zna( Y, Z).`

Reguła 1 na relację `znajomy( X, Y)` opisuje zależność, że jak się kogoś zna, to się jest jego znajomym.
Reguła 2 na `znajomy( X, Z)` jest wykonywana rekurencyjnie, co wynika z

algorytmu w głąb. Rekurencja w tym przypadku jest skończona, ponieważ Reguła 1, która zawsze przy wywołaniu rekurencyjnym będzie sprawdzana przed regułą 2 stanowi jasno zdefiniowany warunek końca rekurencji, czyli osiągnięcie postaci, która nie ma dalszych znajomych.
Prześledźmy wnioskowanie przy danych faktach, które definiują bezpośrednie relacje zna między Vesemirem a Geraltem oraz Geraltem a Ciri:

```
zna(vesemir, geralt).
zna(geralt, ciri).
```

Prolog przeprowadzi dowód na: `znajomy( vesemir, ciri)` w następującej kolejności:

1. Reguła 1: Sprawdza, czy Vesemir zna Ciri. Wynik: fail (nie potwierdzono na podstawie bazy wiedzy
2. Reguła 2: Sprawdza, czy Vesemir jest znajomym kogoś, kto zna Ciri. Sprawdzenie wymaga następujących kroków:
   (a) udowodnienia: `znajomy( vesemir, Y)`:
      i. stosując Regułę 1: Sprawdza, czy Vesemir kogoś zna `zna(vesemir, Y)`
      ii. w liście faktów jest `zna(vesemir, geralt)`, co sprawia, że przy substytucji (Y=geralt) Reguła 1 uzyskuje status TRUE.
   (b) udowodnić drugą część koniunkcji w Regule 2: `zna( Y, ciri)` przy danej liście substytucji (Y=geralt), czyli `zna( geralt, ciri)`. Na podstawie faktów z bazy wiedzy uzyskujemy status TRUE.

   Reguła 2, która jest logicznym iloczynem ma status TRUE dla obu predykatów, co sprawia, że reguła została uzyskała potwierdzona pozytywnie. Tak więc, Vesemir jest znajomym Geralta, a Geralt zna Ciri.

Przykład rekurencji nieskończonej

Zamieńmy kolejnością definicje z przykładu rekurencji skończonej

(Reguła 1:) `znajomy( X, Z) :- znajomy( X, Y), zna( Y, Z).`
`(Reguła 2:) znajomy( X, Y) :- zna( X, Y).`

Przy danych tych samych faktach dowodzimy prawdziwości tego samego zdania. Wnioskowanie przebiega następująco:

1. Reguła 1: Sprawdź, czy Vesemir jest znajomym kogoś (Y), kto zna Ciri. Co wymaga:
   (a) udowodnienia: `znajomy( vesemir, Y)`. Dowód przeprowadzamy wykorzystując definicję na predykat `znajomy`:
      i. stosując Regułę 1: Sprawdź, czy Vesemir jest znajomym kogoś

(Y'), kto zna Y `znajomy( vesemir, Y')`

A. i znów następuję odwołanie do Reguły 1: Sprawdź, czy Vesemir jest znajomym kogoś (Y''), kto zna Y' `znajomy( vesemir, Y'')`

B. i tak dalej w nieskończonym rozwinięciu.

Przedstawiony błąd wynika z niewłaściwej kolejności klauzul. Pierwsza powinna być reguła nierekurencyjna.

– Prolog ma wbudowane funkcje arytmetyczne, nie wymagają one udowodniania. Czyli uniwersum liczb i ich aksjomaty są znane.
– Prolog pozwala na zmianę bazy wiedzy w trakcie wnioskowania. W logice klasycznej nie jest to dopuszczalne. Może to powodować zmianę wnioskowania, np. przez dodanie faktu, który zmieni gałąź dowodową kończącą się niepowodzeniem na kończącą się sukcesem.
– Standardowa unifikacja w Prologu nie wykonuje sprawdzenia, czy zmienna występuje wewnątrz predykatu, z którą jest unifikowana. Może to prowadzić do tworzenia nieskończonych struktur i rekursji. Proces unifikacji może być trudny do śledzenia i zrozumienia, szczególnie dla początkujących programistów. Błędy w unifikacji mogą prowadzić do subtelnych błędów w logice programu, które są trudne do zidentyfikowania i naprawienia.

10.4 Przykłady programów

Języki programowania w logice reprezentują paradygmat deklaratywny, który charakteryzują się tym, że opisują wiedzę a nie algorytm. Celem przybliżenia ich specyfika zostaną przedstawione wybrane przykłady kodów pochodzących z podręczników [Bra00; RN09].

Algorytm w głąb

Jest to szkielet do użycia algorytmu przeszukiwania w głąb podany w języku Prolog. Posługujemy się predykatem `goal`, gdzie za X trzeba podać stan końcowy rozwiązywanego przeszukiwaniem problemu. Predykat `successor(X, S)` musi zostać zdefiniowany i opisywać funkcję generującą kolejne stany (kolejnego potomka) [RN09]. Rekurencyjna procedura `dfs(X)` pobiera jako argument stan problemu w wybranej formie.

```
dfs(X) :- goal(X).
dfs(X) :- successor(X,S),dfs(S).
```

Rozwiązanie problemu 8-hetmanów

Problem n-hetmanów jest zagadką logiczną, poszukującą takiego układu hetmanów na szachownicy o wymiarach $n \times n$, że nie występują pomiędzy bierkami konflikty. Przykład kodu do rozwiązania problemu 8-hetmanów (jedno z wielu rozwiązań z [Bra00]).

```
% solution( BoardPosition) jeżeli
%   BoardPosition jest listą nieatakujących się hetmanów

solution( [] ).

% Pierwszy hetman na X/Y, pozostałe współrzędne w liście Others
solution( [X/Y | Others] )  :-
  solution( Others),
  member( Y, [1,2,3,4,5,6,7,8] ),   % member - predykt wbudowany
  noattack( X/Y, Others).           % 1-szy hetman nie atakuje pozostałych

noattack( _, [] ).                  % Nie ma co atakować

noattack( X/Y, [X1/Y1 | Others] )  :-
  Y =\= Y1,                         % Różne współrzędne Y
  Y1-Y =\= X1-X,                    % Różne przekątne
  Y1-Y =\= X-X1,
  noattack( X/Y, Others).

% Wzorzec wspórzędnych do rozwiązania.
% Współrzędne X są podane kolejno jako 1/, 2/, ..., 8/

template( [1/Y1,2/Y2,3/Y3,4/Y4,5/Y5,6/Y6,7/Y7,8/Y8] ).
```

10.5 Ćwiczenia laboratoryjne (Prolog)

Ćwiczenie 10.1 Ćwiczenie demonstruje podstawy składni języka Prolog:

1. Definiowanie Faktów
   Cel: Nauczyć się definiować fakty w Prologu.
   Zadanie: Zdefiniuj fakty opisujące relacje rodzinne. Na przykład: Adam jest rodzicem Alicji.

2. Proste Zapytania
   Cel: Nauczyć się wykonywać proste zapytania.
   Zadanie: Zapytaj Prologa, kto jest rodzicem Alicji. Zapytaj kto jest matką Alicji.

3. Definiowanie Reguł
   Cel: Nauczyć się tworzyć proste reguły.
   Zadanie:Zdefiniuj regułę określającą, kto jest matką, ojcem, siostrą itp.

4. Rekurencja
   Cel: Zrozumienie podstaw rekurencji w Prologu.
   Zadanie: Zdefiniuj regułę określającą, czy jedna osoba jest przodkiem innej.

5. Lista
   Cel: Nauczyć się pracy z listami w Prologu.
   Zadanie: Napisz regułę sprawdzającą, czy element znajduje się na liście.

6. Negacja
   Cel: Zrozumienie, jak działa negacja w Prologu.
   Zadanie: Sprawdź, czy dana osoba nie jest rodzicem innej osoby.

V Dodatki, spisy

11. Biblioteka *AISearch*

11.1 Wskazówki ogólne

Nie należy modyfikować plików znajdujących się w projekcie `AISearch`. Rozwiązanie zawiera w sobie następujące klasy:

- `IState.cs` — interfejs zawierający w sobie deklaracje metod i właściwości które później będą używane przez klasy `*Search.cs`.
- `State.cs` — klasa abstrakcyjna dziedzicząca po interfejsie `IState.cs`, zawierająca w sobie częściową implementację metod i właściwości używanych przez klasy `*Search.cs`.
- `AlphaBetaSearch.cs` — klasa abstrakcyjna implementująca algorytm *Przycinanie alfa-beta*. W trakcie działania operuje na stanach implementujących interfejs `IState.cs`.
- `AStarSearch.cs` — klasa abstrakcyjna implementująca algorytm A*. W trakcie działania operuje na stanach implementujących interfejs `IState.cs`.
- `BestFirstSearch.cs` — klasa abstrakcyjna implementująca algorytm Best-first search. W trakcie działania operuje na stanach implementujących interfejs `IState.cs`.
- `DepthFirstSearch.cs` — klasa abstrakcyjna implementująca algorytm Depth-first search. W trakcie działania operuje na stanach implementujących interfejs `IState.cs`.
- `PriorityQueue.cs` — implementacja kolejki priorytetowej na kopcu binarnym pozwalająca na szybkie sprawdzenie, czy dany element istnieje, oraz aktualizację elementu. Używana w klasie `AStarSearch.cs`.
- `SimplePriorityQueue.cs` — implementacja kolejki priorytetowej na kopcu binar-

nym pozwalająca na szybkie sprawdzenie, czy dany element istnieje. Używana w klasie `BestFirstSearch.cs`.

Exercise

`Exercise` jest przykładowym projektem, w którym można zaimplementować ćwiczenia laboratoryjne. W razie potrzeby można dołączyć kolejne projekty. Należy jednak pamiętać, aby w nowo utworzonym projekcie dodać referencję do biblioteki `AISearch`. Można to zrobić poprzez kliknięcie PPM na *Odwołania* / *References* w nowo utworzonym Projekcie. Aby w nowo utworzonym pliku skorzystać z klas znajdujących się w bibliotece `AISearch`, należy albo odwoływać się do pełnej nazwy:

```
AISearch.AlfaBetaSearch alfaBeta = new ...
```

albo użyć dyrektywy `using` przed deklaracją przestrzeni nazw:

```
using System;
using AISearch;
namespace MyNamespace {
  ...
}
```

11.2 Wskazówki do implementacji przeszukiwań grafowych

SudokuState.cs

Nowo utworzoną pustą klasę należy zdefiniować jako publiczną i dziedziczącą po klasie bazowej. Klasa powinna być napisana generycznie, czyli powinna pozwalać na reprezentację planszy sudoku dowolnych rozmiarów $n^2 \times n^2$ (domyślnie $n = 3$) oraz zawierać tablicę reprezentującą stan sudoku:

```
using System;
...
using AISearch;
public class SudokuState : State {
    private readonly int n = 3;
    private int[,] table;
    private int GridLength {
        get { return this.n * this.n; }
    }
    public int[,] Table {
        get { return this.table; }
    }
    ...
}
```

Właściwość ID Właściwość `ID` jest właściwością abstrakcyjną dziedziczoną po klasie bazowej i ma na celu zwrócenie stringu jednoznacznie identyfikującego konkretny stan planszy sudoku. Nie powinno dochodzić do konfliktów, czyli dwa odmienne stany powinny posiadać dwie różne wartości `ID`, natomiast dwa stany reprezentujące ten sam

układ na planszy, ale będące dwoma różnymi instancjami klasy, powinny zwracać tę samą wartość `ID`. Proponuje się zaimplementowanie właściwości w następujący sposób:

```
public class SudokuState : State {
    ...
    private string id;
    public override string ID {
        get { return this.id; }
    }
    ...
}
```

gdzie `private string id` jest polem klasy inicjalizowanym w konstruktorze. O właściwościach można poczytać w dokumentacji: `https://docs.microsoft.com/pl-pl/dotnet/csharp/programming-guide/classes-and-structs/properties`.

Metoda ComputeHeuristicGrade Metoda jest metodą abstrakcyjną dziedziczoną po klasie bazowej i ma na celu zwrócenie wartości heurystyki dla konkretnego stanu sudoku. Aby umożliwić kompilację w początkowej fazie prac, można dodać jedynie pustą definicję metody:

```
public class SudokuState : State {
    ...
    public override double ComputeHeuristicGrade() {
        throw new NotImplementedException();
    }
    ...
}
```

Docelowo metoda powinna liczyć heurystykę „liczba niewiadomych”.

Metoda Print W klasie należy również dodać metodę `Print` wyświetlającą stan sudoku na ekranie. Bezwzględnie wymaga się, aby metoda wyświetlała stan w postaci macierzy 9x9 w czytelny sposób, tj. wszystkie puste pola (zawierające 0) muszą być zamieniane na spacje, ewentualnie nowo wstawiany stan powinien być wyświetlany innym kolorem. Metoda powinna rysować linie oddzielające małe kwadraty $n \times n$. W razie potrzeby należy zwiększyć bufor konsoli, aby wszystkie możliwe stany mogły zostać wyświetlone. Można zrobić to ręcznie w ustawieniach konsoli albo umieszczając w metodzie `Main` w klasie `Program` polecenie:

```
Console.BufferHeight = 1000;
```

Do współpracy z konsolą służy klasa `Console`: `https://docs.microsoft.com/pl-pl/dotnet/api/system.console`.

Konstruktory W klasie nie może zabraknąć konstruktorów. Do poprawnej implementacji potrzebne będą dwa konstruktory. Pierwszy przyjmujący string np. postaci

"00070080000004003..." reprezentujący początkowy stan sudoku. Drugi konstruktor jest odpowiedzialny za utworzenie potomka stanu podanego w parametrze o wartościach podanych w parametrze:

```
public SudokuState(int n, string sudokuPattern) : base() {
    this.n = n;

    if (sudokuPattern.Length != GridLength * GridLength) {
        throw new ArgumentException("Niewłaściwa długość
            sudokuPattern.");
    }

    this.id = sudokuPattern;
    this.table = new int[GridLength, GridLength];

    for(int i = 0; i < GridLength; ++i) {
        for(int j = 0; j < GridLength; ++j) {
            this.table[i, j] = sudokuPattern[i * GridLength
                + j] - 48;
        }
    }
    //obliczenie heurystyki
    this.h = ComputeHeuristicGrade();
}

public SudokuState(SudokuState parent, int newValue, int x,
    int y) : base(parent) {
    this.table = new int[GridLength, GridLength];

    //skopiowanie stanu sudoku do nowej tabeli
    Array.Copy(parent.table, this.table, this.table.Length);

    //ustawienie nowej wartości w wybranym polu sudoku
    this.table[x, y] = newValue;

    //utworzenie nowego id odpowiadającemu aktualnemu
        stanowi planszy
    StringBuilder builder = new StringBuilder(parent.id);
    builder[x*GridLength + y] = (char)(newValue + 48);
    this.id = builder.ToString();

    this.h = ComputeHeuristicGrade();
}
```

Części kodu `:base()` i `:base(parent)` są odpowiedzialne za wywołanie konstruktora z klasy bazowej. Klasa `StringBuilder` pozwala na efektywniejsze zarządzanie zasobami komputera podczas pracy z ciągami znakowymi.

SudokuSearch.cs

Nowo utworzoną pustą klasę należy zdefiniować jako publiczną i dziedziczącą po klasie bazowej. W klasie wystarczy zdefiniować pusty konstruktor oraz dwie metody abstrakcyjne. Metoda `isSolution` zwraca informację, czy dany stan jest stanem końcowym. Metoda `buildChildren` ma za zadanie zbudowanie potomków wybranego stanu. Poniżej przestawiona jest podstawowa wersja metody:

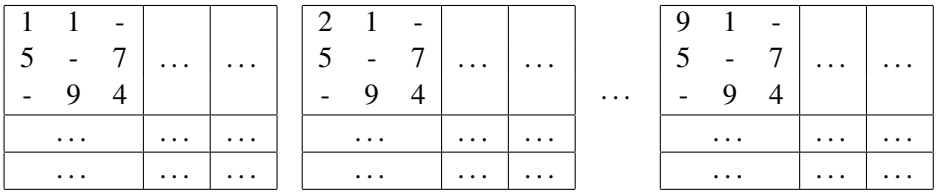
```
...
public class SudokuSearch : BestFirstSearch {
    public SudokuSearch(SudokuState state) : base(state) { }
    protected override void buildChildren(IState parent) {
        SudokuState state = (SudokuState)parent;
        //poszukiwanie wolnego pola
        for (int i = 0; i < SudokuState.GRID_SIZE; ++i) {
            for (int j = 0; j < SudokuState.GRID_SIZE; ++j){
                if (state.Table[i, j] == 0) {
                    //wstawianie potomków w wolne pole
                    for (int k = 1; k <
                        SudokuState.GRID_SIZE + 1; ++k) {
                        SudokuState child = new
                            SudokuState(state, k, i, j);
                        parent.Children.Add(child);
                    }
                    return;
                }
            }
        }
    }
    protected override bool isSolution(IState state) {
        return state.H == 0.0;
    }
}
```

Rozważmy sobie stan reprezentujący planszę sudoku, która ma następującą postać:

- 1 - 5 - 7 - 9 4	...	...
...	...	...
...	...	...

Jako stany potomne ww. planszy rozumiemy następujące plansze sudoku:

Należy mieć na uwadze, że w ww. przykładzie nie wszyscy utworzeni potomkowie będą poprawnymi stanami gry sudoku. W niektórych z nich nowo wstawiona cyfra w polu *(i, j)* będzie już występowała w danym wierszu, kolumnie lub małym kwadracie. Tego typu stanom metoda `ComputeHeuristicGrade` powinna nadać wartość heurystyki równą $+\infty$ (`double.PositiveInfinity`). Potomków można generować w oparciu o inne pole *(i, j)*, a w przypadku bardziej skomplikowanych heurystyk jest to wymagane. **Uwaga!**



potomków zawsze podpinamy w jedno puste pole, dlatego każdy ze stanów będzie posiadał maksymalnie tylko 9 potomków.

Program.cs

Klasa `Program.cs` zawiera metodę `Main`. W niej należy wyświetlić kolejne stany Sudoku prowadzące do rozwiązania. Można zrobić to w następujący sposób:

```
static void Main(string[] args) {
    //sudoku powinno zawierać 81 cyfr
    string sudokuPattern = "010330218...";

    SudokuState startState = new SudokuState(sudokuPattern);
    SudokuSearch searcher = new SudokuSearch(startState);
    searcher.DoSearch();

    IState state = searcher.Solutions[0];
    List<SudokuState> solutionPath = new
        List<SudokuState>();

    while (state != null) {
        solutionPath.Add((SudokuState)state);
        state = state.Parent;
    }

    solutionPath.Reverse();
    foreach(SudokuState s in solutionPath){
        s.Print();
    }
}
```

11.3 Wskazówki do implementacji gry Connect4

Connect4State.cs

Nowo utworzoną pustą klasę należy zdefiniować jako publiczną i dziedziczącą po klasie bazowej. Należy zaimplementować właściwość `ID`, która ma na celu zwrócenie stringu jednoznacznie identyfikującego konkretny stan planszy. Nie powinno dochodzić do konfliktów, czyli dwa odmienne stany powinny posiadać dwie różne wartości `ID`, natomiast dwa stany reprezentujące ten sam układ na planszy, ale będące dwoma różnymi instancjami klasy, powinny zwracać tę samą wartość `ID`.

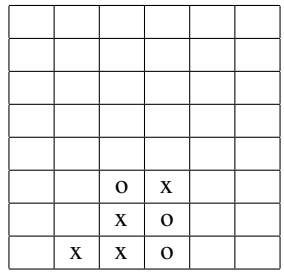
Konstruktory Do poprawnej implementacji potrzebne będą dwa konstruktory. Pierwszy tworzący pustą planszę Connect4. Drugi konstruktor jest odpowiedzialny za utworzenie potomka stanu podanego w parametrze o wartościach podanych w parametrze. Poniżej przedstawiono fragmenty kodu wymagane w drugim konstruktorze, które są niezbędne do poprawnego działania programu.

```
public Connect4State(Connect4State parent, ... /*pozostałe
    niezbędne parametry*/) : base(parent) {
    //reszta implementacji

    //ustawienie stringu identyfikującego stan.
    this.id = ...
    //ustawienie, na ktorym poziomie w drzewie znajduje sie
        stan.
    this.depth = parent.depth + 0.5;

    //Bardzo wazne. Nie ustawiany na czubek drzewa, z
        którego budujemy stany, tylko na pierwsze pokolenie
        stanów potomnych.
    if (parent.rootMove == null) {
        this.rootMove = this.id;
    }
    else {
        this.rootMove = parent.rootMove;
    }
    //Dodanie stanu do potomków stanu rodzicielskiego
    parent.Children.Add(this);
}
```


ComputeHeuristicGrade i przykładowa heurystyka Metoda ma na celu zwrócenie wartości heurystyki dla konkretnego stanu planszy Connect4. Przykładowa heurystyka może mieć następującą postać. Pojedynczy stan jest punktowany za 1 punkt, dwa stany z rzędu (w pionie, w poziomie i po skosie) jako 4 punkty, 3 stany z rzędu jako 16 punktów.

liczba stanów pod rząd	gracz maksymalizujący	gracz minimalizujący
1 stan	1	-1
2 stany	4	-4
3 stany	16	-16
4 stany	∞	$-\infty$

Rozważając przykładową planszę Connect4:



i zakładając, że „x” jest graczem maksymalizującym a „o” minimalizującym, planszę możemy ocenić w następujący sposób: gracz Max zbierze 24 punkty (2 dwójki i 1 trójka), a gracz Min zbierze -8 punktów (2 dwójki). Dlatego ocena heurystyczna tego konkretnego stanu planszy wynosi $24 - 8 = 16$, czyli przewagę posiada gracz maksymalizujący. **Jest to jedynie propozycja heurystyki** — student może i powinien zaproponować własną. W finalnej heurystyce można uwzględniać następujące rzeczy: preferowanie centrum niż boków (lub odwrotnie), bliskość do sufitu, itd. **Uwaga!** w pierwszej kolejności upewnij się, że metoda zwraca odpowiednie „nieskończoności” (`double.PositiveInfinity`, `double.NegativeInfinity`) dla stanów zwycięskich.

Connect4Search.cs

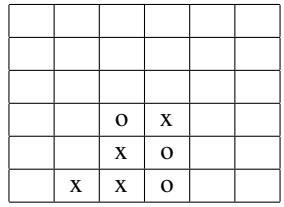
Nowo utworzoną pustą klasę należy zdefiniować jako publiczną i dziedziczącą po klasie bazowej oraz zaimplementować metody abstrakcyjne. W klasie wystarczy zdefiniować pusty konstruktor:

```
public Connect4Search(IState startState, bool
    isMaximizingPlayerFirst, int maximumDepth) :
    base(startState, isMaximizingPlayerFirst, maximumDepth)
    { }
```

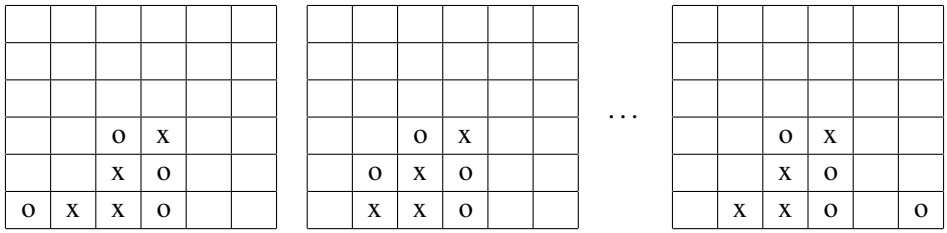
Parametry są następujące:

- **startState** — wybrany stan planszy Connect4, dla której chcemy wykonać algorytm alfa-beta w celu znalezienia kolejnego ruchu,
- **isMaximizingPlayerFirst** — określa, który z graczy zaczyna rozgrywkę,
- **maximumDepth** — definiuje głębokość przeszukiwania w drzewie.

buildChildren Metoda ma za zadanie zbudowanie potomków wybranego stanu. Rozważając następujący stan:



i zakładając, że kolejny jest ruch gracza „o”, stany potomne będą miały postać:



MovesMiniMaxes Jest to właściwość, która po każdym wywołaniu metody `DoSearch` zawiera w sobie stany zwrócone przez algorytm wraz z wartością przypisanej im heurystyki. Ze zbioru należy wybrać stan o największej wartości heurystyki w przypadku gracza Max lub najmniejszej wartości heurystyki w przypadku gracza Min. W celu przeszukania kolekcji można posłużyć się pętlą `foreach`:

```
foreach (KeyValuePair<string, double> kvp in
    this.MovesMiniMaxes) {
    //...
}
```

Program.cs

Klasa `Program.cs` zawiera metodę `Main`. W metodzie należy zaimplementować rozgrywkę pomiędzy człowiekiem a sztuczną inteligencją. Rozgrywka powinna odbywać się według następującego schematu:

1. po wykonaniu ruchu przez człowieka stan powinien zostać podany w konstruktorze klasy `Connect4Search`,
2. wywołanie metody `DoSearch` (działania algorytmu alfa-beta),
3. wybranie najlepszego stanu z właściwości `MovesMiniMaxes` oraz wyświetlenie go użytkownikowi,
4. wykonanie ruchu przez użytkownika.

Oczywiście po każdym ruchu należy sprawdzać, czy dany gracz nie wygrał. W przypadku wykorzystania stanu w kolejnym ruchu jako stanu początkowego, należy pamiętać o tym, aby wyczyścić pola `this.parent`, `this.rootMove` oraz `this.depth`.

Bibliografia

Źródła drukowane

[AB09] M. Anthony i P.L. Bartlett. *Neural Network Learning: Theoretical Foundations*. Cambridge, UK: Cambridge University Press, 2009.

[Bar98] P.L. Bartlett. „The sample complexity of pattern classification with neural networks: the size of weights is more important than the size of the network". W: *IEEE Transactions on Information Theory* 44.2 (1998), s. 525–536.

[Ber18] Michael K. Bergman. „Information, Knowledge, Representation". W: *A Knowledge Representation Practionary: Guidelines Based on Charles Sanders Peirce*. Cham: Springer International Publishing, 2018, s. 15–42. ISBN: 978-3-319-98092-8.

[BR98] J. Bilski i L. Rutkowski. „A fast training algorithm for neural networks". W: *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing* 45.6 (lip. 1998), s. 749–753. ISSN: 1057-7130. DOI: 10.1109/82.686696.

[BŚ00] Jarosław Bilski i Adriana Świąć. „Metoda wstecznej propagacji błędów i jej modyfikacje". W: red. Włodzisław Duch i in. Biocybernetyka

i inżynieria biomedyczna 2000 3. Warszawa: Akademicka Oficyna Wydawnicza EXIT, 2000, s. 73–109. ISBN: 83–87674–18–4.

[BL04] Ronald J. Brachman i Hector J. Levesque. *Knowledge Representation and Reasoning*. Elsevier, 2004, s. I–XXIX, 1–381. ISBN: 978-1-55860-932-7.

[Bra00] Ivan Bratko. *Prolog Programming for Artificial Intelligence*. 3 wyd. Harlow, England: Pearson Addison-Wesley, 2000. ISBN: 978-0-201-40375-6.

[Bru63] A. Brudno. „Bounds and valuations for shortening the search of estimates". W: *Problems of Cybernetics (Problemy Kibernetiki)* 10 (1963), s. 225–241.

[CRS94] S. Chari, P. Rohatgi i A. Srinivasan. „Improved algorithms via approximations of probability distributions". W: *Proceedings of the Twenty-Sixth Annual ACM Symposium on the Theory of Computing*. 1994, s. 584–592.

[CM07] V. Cherkassky i F. Mulier. *Learning from Data*. 2 wyd. USA: John Wiley & Sons, inc., 2007.

[CL75] Allan M Collins i Elizabeth F Loftus. „A spreading-activation theory of semantic processing." W: *Psychological review* 82.6 (1975), s. 407.

[Cyb89] G. Cybenko. „Approximation by superpositions of a sigmoidal function". W: *Mathematics of Control, Signals, and Systems (MCSS)* 2.4 (grud. 1989), s. 303–314. ISSN: 0932-4194. DOI: 10.1007/BF02551274. URL: http://dx.doi.org/10.1007/BF02551274.

[Dav91] W.C. Davidon. „Variable metric method for minimization". W: *SIAM Journal on Optimization* 1.1 (1991), s. 1–17.

[Dav85] Lawrence Davis. „Applying Adaptive Algorithms to Epistatic Domains". W: *Proceedings of the 9th International Joint Conference on Artificial Intelligence - Volume 1*. IJCAI'85. Los Angeles, California: Morgan Kaufmann Publishers Inc., 1985, s. 162–164. ISBN: 0934613028.

[Dij59] E.W. Dijkstra. „A note on two problems in connexion with graphs". W: *Numerische Mathematik* 1.1 (1959), s. 269–271. ISSN: 0029-599X. DOI: {10.1007/BF01386390}.

[Doz16] T. Dozat. „Incorporating Nesterov Momentum into Adam". W: *Proc. of 4th Int. Conf. on Learning Representations (ICLR)*. 2016, s. 1–4.

[DHS11] J. Duchi, E. Hazan i Y. Singer. „Adaptive Subgradient Methods for Online Learning and Stochastic Optimization”. W: *Journal of Machine Learning Research* 12.61 (2011), s. 2121–2159. URL: http://jmlr.org/papers/v12/duchi11a.html.

[EH63] D. Edwards i T. Hart. *The Alpha-Beta Heuristic*. Spraw. tech. 30. Massachusetts Institute of Technology, 1963.

[Ger99] Neil Gershenfeld. *The Nature of Mathematical Modeling*. Cambridge, United Kingdom: Cambridge University Press, 1999.

[GB10] X. Glorot i Y. Bengio. „Understanding the difficulty of training deep feedforward neural networks”. W: *Journal of Machine Learning Research — Proceedings Track* 9 (sty. 2010), s. 249–256.

[GL85] D.E. Goldberg i R. Lingle. „Alleles, loci, and the traveling salesman problem”. W: *Proceedings of an international conference on genetic algorithms and their applications*. 1985, s. 154–159.

[Gol89] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. 1st. USA: Addison-Wesley Longman Publishing Co., Inc., 1989. ISBN: 0201157675.

[Gwi07] T.D. Gwiazda. *Algorytmy genetyczne: Kompendium TOM 2*. Wydawnictwo Naukowe PWN, 2007. ISBN: 9788301153816.

[Gwi09] T.D. Gwiazda. *Algorytmy genetyczne: Kompendium TOM 1*. Wydawnictwo Naukowe PWN, 2009. ISBN: 9788301158309.

[HM94] M. T. Hagan i M. B. Menhaj. „Training Feedforward Networks with the Marquardt Algorithm”. W: *Trans. Neur. Netw.* 5.6 (list. 1994), s. 989–993. ISSN: 1045-9227. DOI: 10.1109/72.329697. URL: http://dx.doi.org/10.1109/72.329697.

[HMY85] O. Hansson, A.E. Mayer i M.M. Yung. *Generating Admissible Heuristics by Criticizing Solutions to Relaxed Models*. Spraw. tech. CUCS-219-85. New York, N.Y., 10027, USA: Department of Computer Science, Columbia University, 1985.

[HLP08] Frank van Harmelen, Vladimir Lifschitz i Bruce Porter, red. *Handbook of Knowledge Representation*. T. 3. Foundations of Artificial Intelligence. Elsevier, 2008. ISBN: 978-0-444-52211-5.

[HNR68] P.E. Hart, N.J. Nilsson i B. Raphael. „A Formal Basis for the Heuristic Determination of Minimum Cost Paths”. W: *IEEE Transactions on Systems Science and Cybernetics* 4.2 (1968), s. 100–107.

[HNR72] P.E. Hart, N.J. Nilsson i B. Raphael. „Correction to “A Formal Basis for the Heuristic Determination of Minimum Cost Paths””. W: *SIGART Bull.* 37 (1972), s. 28–29.

[Hau95] D. Haussler. „Sphere packing numbers for subsets of the Boolean *n*-cube with bounded Vapnik-Chervonenkis dimension”. W: *Journal of Combinatorial Theory, Series A* 69.2 (1995), s. 217–232.

[HL95] D. Haussler i P.M. Long. „A generalization of Sauer’s lemma”. W: *Journal of Combinatorial Theory, Series A* 71.2 (1995), s. 219–240.

[HSS12] G. Hinton, N. Srivastava i K. Swersky. *Overview of mini-batch gradient descent*. (wykład 6, praca nieopublikowana, data dostępu: 10.07.2022). 2012. URL: `http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf`.

[Hol75] John H. Holland. *Adaptation in Natural and Artificial Systems*. second edition, 1992. Ann Arbor, MI: University of Michigan Press, 1975.

[HJ61] R. Hooke i T.A. Jeeves. „“Direct search” solution of numerical and statistical problems”. W: *Journal of ACM* 8.2 (1961), s. 212–229.

[KE95] J. Kennedy i R. Eberhart. „Particle Swarm Optimization”. W: *Proceedings of IEEE International Conference on Neural Networks*. T. IV. 1995, s. 1942–1948.

[KB14] D.P. Kingma i J. Ba. „Adam: A Method for Stochastic Optimization”. W: *ICLR (Poster)*. 2014. URL: `https://arxiv.org/pdf/1412.6980.pdf`.

[Klę05] P. Klęsk. „Metoda nadawania pożądanych własności ekstrapolacyjnych neuronowym i rozmytym modelom systemów wielowymiarowych”. praca doktorska. Politechnika Szczecińska, Wydział Informatyki, 2005.

[Klę12] P. Klęsk. „Zdolność do uogólniania w uczeniu maszynowym”. praca habilitacyja. Zachodniopomorski Uniwersytet Technologiczny w Szczecinie, Wydział Informatyki, 2012.

[KK11] P. Klęsk i M. Korzeń. „Sets of approximating functions with finite Vapnik–Chervonenkis dimension for nearest-neighbors algorithms”. W: *Pattern Recognition Letters* 32.14 (2011), s. 1882–1893.

[KM75] D.E. Knuth i R.W. Moore. „An analysis of alpha-beta pruning”. W: *Artificial Intelligence* 6.4 (1975), s. 293–326.

[Kol57] A. K. Kolmogorov. „On the Representation of Continuous Functions of Several Variables by Superposition of Continuous Functions of One Variable and Addition". W: *Doklady Akademii Nauk SSSR* 114 (1957), s. 369–373.

[Kor85] R.E. Korf. „Depth-first Iterative-Deepening: An Optimal Admissible Tree Search". W: *Artificial Intelligence* 27 (1985), s. 97–109.

[LLS20] Zhiyuan Liu, Yankai Lin i Maosong Sun. *Representation Learning for Natural Language Processing*. Springer Nature Singapore, 2020. DOI: 10.1007/978-981-15-5573-2.

[Met+53] N. Metropolis i in. „Equation of State Calculations by Fast Computing Machines". W: *Journal of Chemical Physics* 21.6 (1953), s. 1087.

[Mur12] Kevin P. Murphy. *Machine Learning: A Probabilistic Perspective*. The MIT Press, 2012. ISBN: 9780262018029.

[Nes83] Y. Nesterov. *A method for unconstrained convex minimization problem with the rate of convergence* $O(1/k^2)$. 1983.

[Neu28] J. von Neumann. „Zur Theorie der Gesellschaftsspiele". W: *Mathematische Annalen* 100.1 (1928), s. 295–320. DOI: 10.1007/BF01448847.

[NM44] J. von Neumann i O. Morgenstern. *Theory of Games and Economic Behavior*. Princeton University Press, 1944.

[NS76] A. Newell i A.H. Simon. „Computer Science and Empirical Inquiry: Symbols and Search". W: *Communnications of the ACM* 19.3 (1976), s. 113–126.

[Nil98] Nils J. Nilsson. *Artificial Intelligence: A New Synthesis*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998. ISBN: 9780080499451.

[Nov62] A.B. Novikoff. „On convergence proofs on perceptrons". W: *Symposium on the Mathematical Theory of Automata*. Polytechnic Institute of Brooklyn, 1962.

[OSH87] I. M. Oliver, D. J. Smith i J. R. C. Holland. „A Study of Permutation Crossover Operators on the Traveling Salesman Problem". W: *Proceedings of the Second International Conference on Genetic Algorithms on Genetic Algorithms and Their Application*. Cambridge, Massachusetts, USA: L. Erlbaum Associates Inc., 1987, s. 224–230. ISBN: 0805801588.

[Oso00] Stanisław Osowski. *Sieci neuronowe do przetwarzania informacji*. I. Warszawa: Biuro Wydawnicze Politechniki of Warszawskiej, 2000. ISBN: 83–7207–187–X.

[Pea82] J. Pearl. „The Solution for the Branching Factor of the Alpha-beta Pruning Algorithm and Its Optimality". W: *Commununications of the ACM* 25.8 (1982), s. 559–564. DOI: 10.1145/358589.358616.

[Pea84] J. Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1984. ISBN: 0-201-05594-5.

[Pie19] Marcin Pietrzykowski. „Lokalne uczenie algorytmów regresyjnych metodą mini-modeli". praca doktorska. Zachodniopomorski Uniwersytet Technologiczny w Szczecinie, 2019.

[Pol64] B. Polyak. „Some methods of speeding up the convergence of iteration methods". W: *USSR Computational Mathematics and Mathematical Physics* 4 (sty. 1964), s. 1–17.

[Red+18] S.J. Reddi i in. „On the Convergence of Adam and Beyond". W: *Proc. of 6th Int. Conf. on Learning Representations (ICLR)*. 2018.

[RM99] Russel D. Reed i Robert J. Marks, II. *Neural Smithing*. London, England: A Bradford Book, The MIT Press, 1999. ISBN: 0–262–18190–8.

[RB93] M. Riedmiller i H. Braun. „A direct adaptive method for faster backpropagation learning: the RPROP algorithm". W: *IEEE International Conference on Neural Networks*. T. 1. Mar. 1993, s. 586–591. DOI: 10.1109/ICNN.1993.298623.

[Ros58] F. Rosenblatt. „The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain". W: *Psychological Review* 65.6 (1958), s. 386–408.

[Rud16] S. Ruder. *An overview of gradient descent optimization algorithms*. data dostępu: 10.07.2022. 2016. URL: https://ruder.io/optimizing-gradient-descent/index.html.

[RHW86] D. E. Rumelhart, G. E. Hinton i R. J. Williams. „Learning Internal Representations by Error Propagation". W: *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1*. Red. David E. Rumelhart, James L. McClelland i CORPORATE PDP Research Group. Cambridge, MA, USA: MIT Press, 1986, s. 318–362. ISBN: 0-262-68053-X. URL: http://dl.acm.org/citation.cfm?id=104279.104293.

[RN09] S. Russell i P. Norvig. *Artificial Intelligence: A Modern Approach*. 3 wyd. Prentice Hall Press, 2009. ISBN: 978-0136042594.

[Rut12] L. Rutkowski. *Metody i techniki sztucznej inteligencji*. Informatyka - Zastosowania. Wydawnictwo Naukowe PWN, 2012. ISBN: 9788301157319.

[Sau72] N. Sauer. „On the density of families of sets". W: *Journal of Combinatorial Theory, Series A* 13 (1972), s. 145–147.

[Shi11] E. Shimon. *Graph Algorithms*. Cambridge University Press, 2011. ISBN: 978-0-521-73653-4.

[Sho05] W. Shortz. *Sudoku Easy*. St. Martin's Griffin, 2005.

[Ste78] J.M. Steel. „Existence of submatrices with all possible columns". W: *Journal of Combinatorial Theory, Series A* 24 (1978), s. 84–88.

[Val84] L.G. Valiant. „A theory of the learnable". W: *Communications of the ACM* 27.11 (1984), s. 1134–1142.

[Vap95] V.N. Vapnik. *The Nature of Statistical Learning Theory*. New York: Springer Verlag, 1995.

[Vap98] V.N. Vapnik. *Statistical Learning Theory: Inference from Small Samples*. New York: Wiley, 1998.

[VC71] V.N. Vapnik i A.J. Chervonenkis. „On the uniform convergence of relative frequencies of events to their probabilities". W: *Theory of Probability and its Applications* 16.2 (1971), s. 264–280.

[Wol96] D. Wolpert. „The Lack of A Priori Distinctions between Learning Algorithms". W: *Neural Computation* 8 (1996), s. 1341–1390.

[Wri15] S.J. Wright. „Coordinate descent algorithms". W: *Mathematical Programming* 151.1 (2015), s. 3–34.

[Zha02] Tong Zhang. „Covering Number Bounds of Certain Regularized Linear Function Classes". W: *Journal of Machine Learning Research* 2 (2002), s. 527–550.

Źródła internetowe

[Wik19] Wikipedia. *Sudoku*. Accessed: 22.12.2019. 2019. URL: `http://en.wikipedia.org/wiki/Sudoku`.

Spis rysunków

Spis tabel

Skorowidz

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

W

Z