Sztuczna inteligencja (draft)

Wydział Informatyki, ZUT w Szczecinie

Przemysław Klęsk
Marcin Pietrzykowski
Joanna Kołodziejczyk
Jacek Klimaszewski

PUBLISHED BY WYDZIAŁ INFORMATYKI

WI.ZUT.EDU.PL

Wydanie pierwsze, październik 2020

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 10.

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...

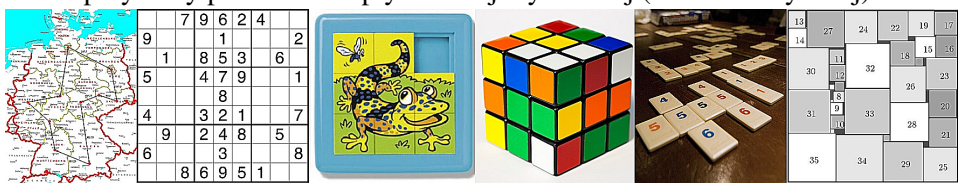
Algorytmy przeszukujące stanowią znaczącą część badań w ramach sztucznej inteligencji. Patrząc z perspektywy historycznej na rozwój tej dziedziny, *znajdowanie ścieżek* i *szachy* przychodzą na myśl jako prawdopodobnie dwa najbardziej naturalne przykłady zadań, gdzie algorytmy przeszukujące stosowano 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*.

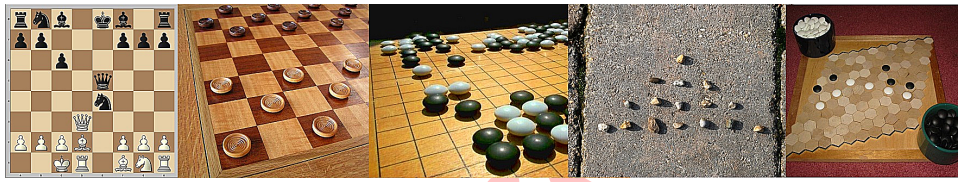
Rys. 1.1 przedstawia ilustracyjnie dwie powyższe grupy.

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 z wykorzystaniem 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, które przeprowadza obiekt w stan o pewnych porządanych 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 czynności (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:

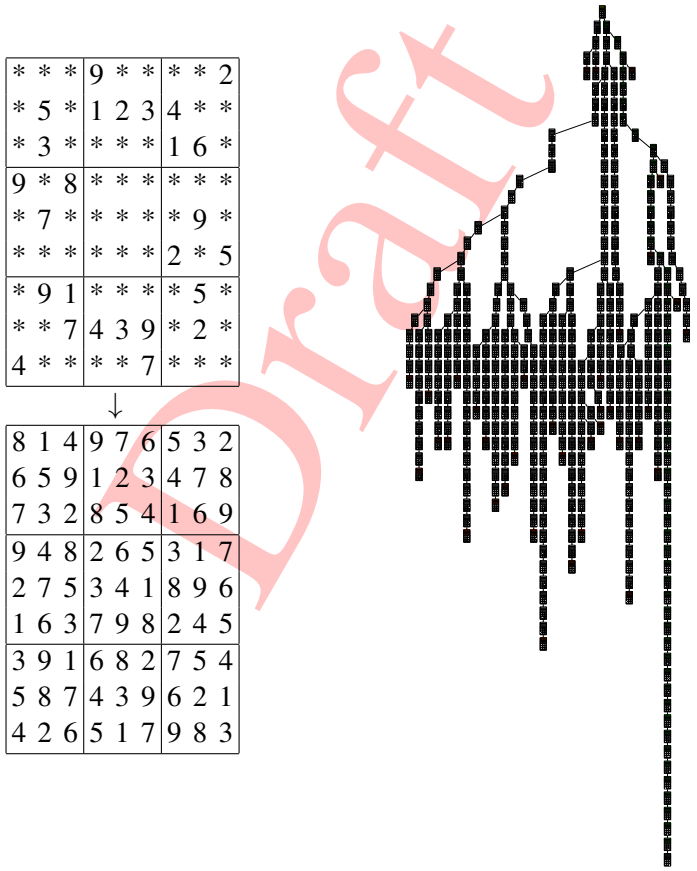
1. **Generowanie stanów potomych** — *Jakie nowe stany (bezpośredni potomkowie) mogą zostać wygenerowane z danego stanu?*
2. **Identyfikacja** — *Jakie identyfikatory (napisowe lub całkowitoliczbowe) mogą zostać przypisane do stanów, tak aby ten sam stan nie był odwiedzany niepotrzebnie wielokrotnie?*
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*

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

(drzewa gier).

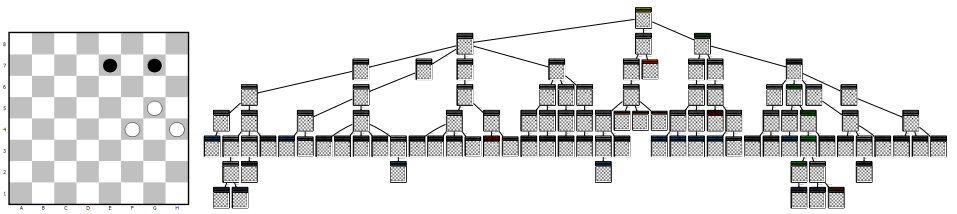
Ostatni element oznaczono numerem 3', ponieważ może on być tak naprawdę postrzegany jako rozszerzenie elementu 3. Mowa w nim o pewnej funkcji oceniającej, nazywanej często funkcją heurystyczną lub krótko heurystyką. W zależności od rodzaju rozpatrywanego zadania przeszukiwania (grafy czy gry) dokładny 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 przykładowe graf i drzewo przeszukiwań wygene-

rowane 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 [47, 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 [10] 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 lub czas podróży. 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.

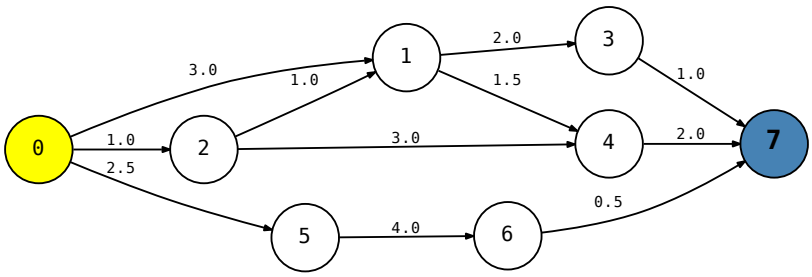
Twierdzenie 2.1.1 Niech s^* oznacza stan docelowy. Jeżeli istnieje przynajmniej jedna ścieżka pomiędzy s_0 a s^*, to algorytm Dijkstry (Algorytm 3) gwarantuje znalezienie ścieżki najkrótszej — tj. ścieżki o najmniejszej wartości kosztu g.

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, że

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

$g(s) \geqslant g(s^*)$. 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:

Breadth-first search (przejście wszerz): $(0,1,2,5,3,4,6,7)$,
Depth-first search (przejście w głąb): $(0,1,3,7)$,
algorytm Dijkstry: $(0,2,1,5,4,3,7)$.

(!) 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. Wreszcie mnożąc (2.1) i (2.2) otrzymujemy liczbę rzędu $1.6 \cdot 10^{46}$.

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 [38] 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 [18].

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 $(7/3, 7 \mod 3) = (2,1)$ — 7-ka powinna leżeć w wierszu nr 2 i kolumnie nr 1 (numerowanie od zera). A zatem wzór heurystyki „Manhattan” można zapisać następująco:

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

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 [18] 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

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

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 [48, 56] 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 pracy [19, 20]. 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. Formalna definicja jest więc następująca:

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
```

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.

Dlaczego monotoniczność heurystyki jest użyteczną własnością? Wyobraźmy sobie, że zetknęliśmy się z pewnym nowym i trudnym problemem optymalizacyj-

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

nym, 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,
(b) lub zwiększyć się o 1, $h(t) = h(s) + 1$, gdy przesuwany kafelek był na właściwym miejscu przed ruchem,
(c) lub 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.

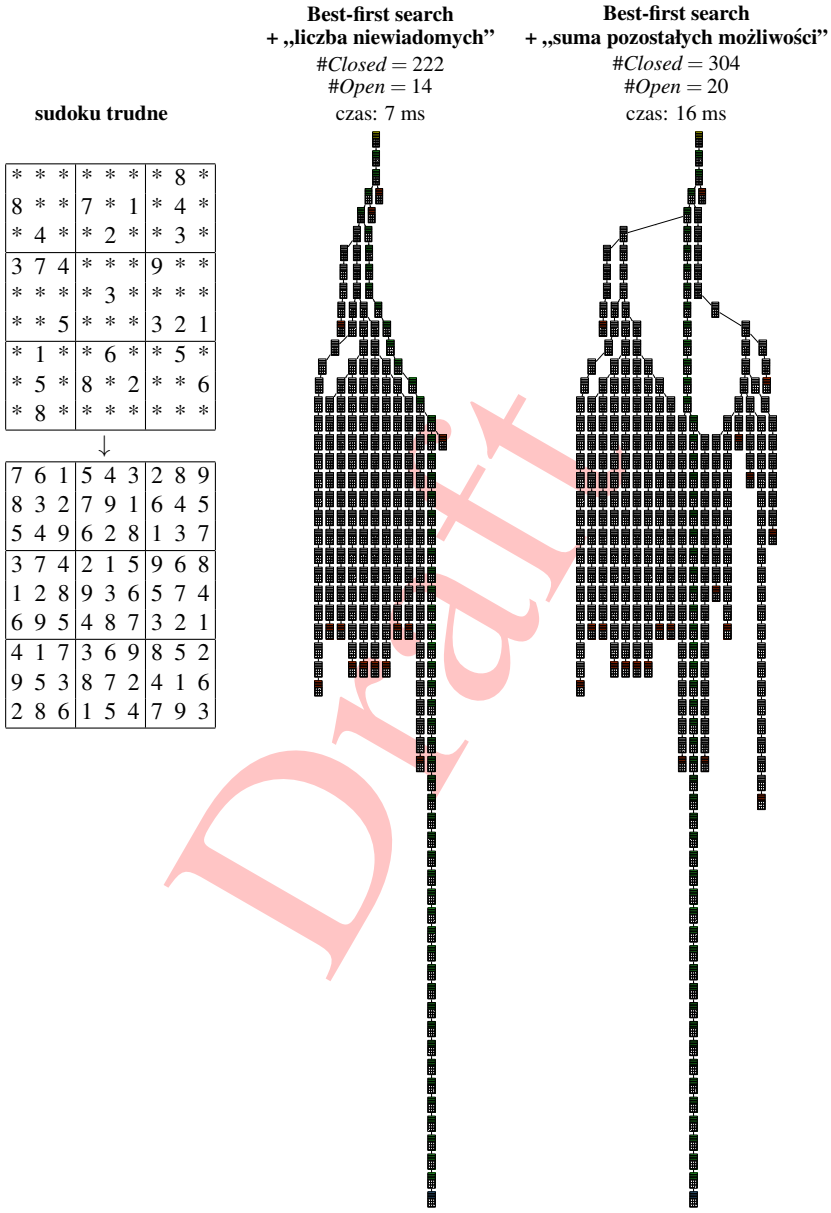
2.3.5 Przykłady działania Best-first search i A*

Sudoku

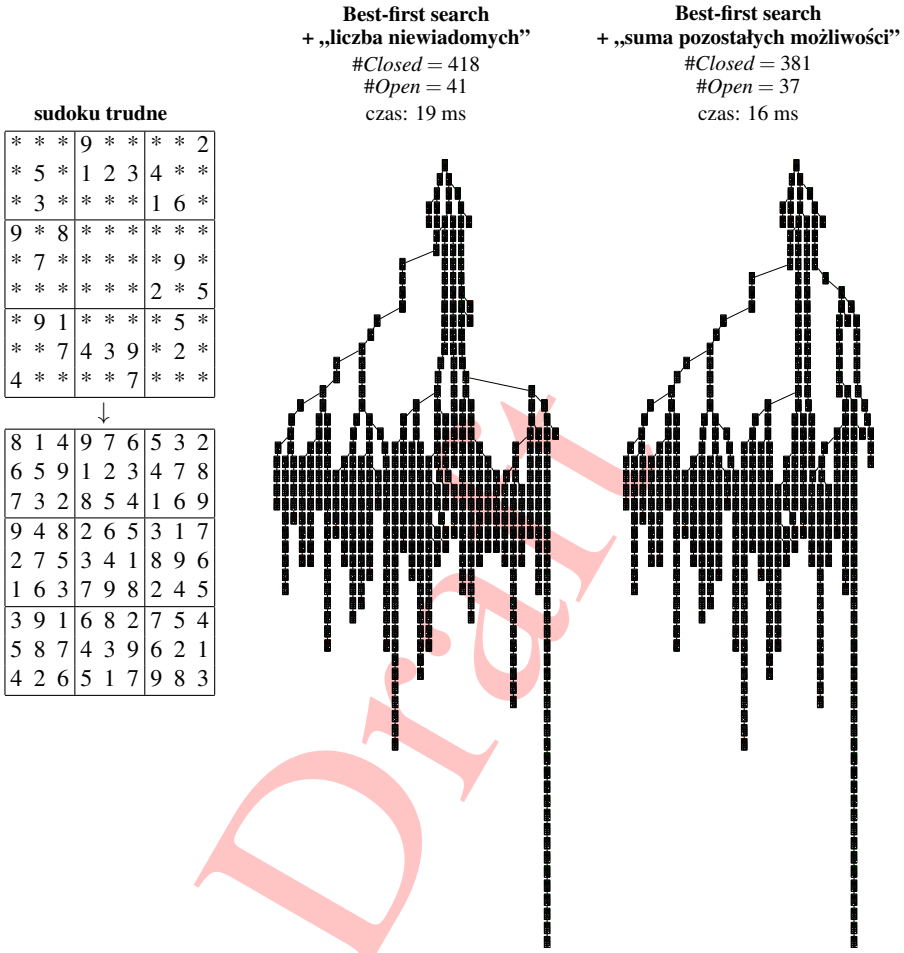
Na rysunkach 2.3, 2.4 przedstawiono grafy przeszukiwań wygenerowane przez algorytm Best-first search z użyciem różnych heurystyk dla dwóch łamigłówek sudoku (określanych jako trudne). Zachęcamy Czytelnika do powiększenia przeglądanych ilustracji. Kolorem żółtym zaznaczono stan początkowy, a niebieskim 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).

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"

[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*).

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.

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

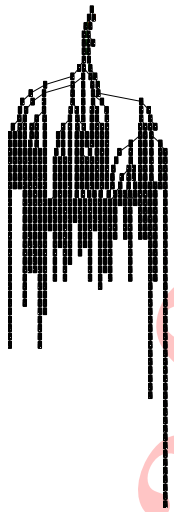
sudoku „Qassim Hamza”
(bardzo trudne)

*	*	*	7	*	*	8	*	*
*	*	*	*	4	*	*	3	*
*	*	*	*	*	9	*	*	1
6	*	*	5	*	*	*	*	*
*	1	*	*	3	*	*	4	*
*	*	5	*	*	1	*	*	7
5	*	*	2	*	*	6	*	*
*	3	*	*	8	*	*	9	*
*	*	7	*	*	*	*	*	2

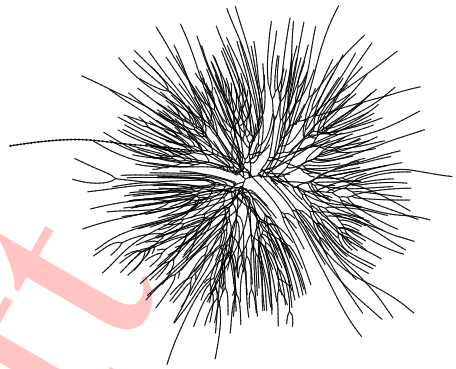
↓

3	2	9	7	1	6	8	5	4
1	7	6	8	4	5	2	3	9
4	5	8	3	2	9	7	6	1
6	4	3	5	7	2	9	1	8
7	1	2	9	3	8	5	4	6
8	9	5	4	6	1	3	2	7
5	8	1	2	9	4	6	7	3
2	3	4	6	8	7	1	9	5
9	6	7	1	5	3	4	8	2

Best-first search
+ „liczba niewiadomych”
#Closed = 525
#Open = 40
czas: 70 ms



Best-first search
+ „suma pozostałych możliwości”
#Closed = 5 267
#Open = 452
czas: 208 ms



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*).

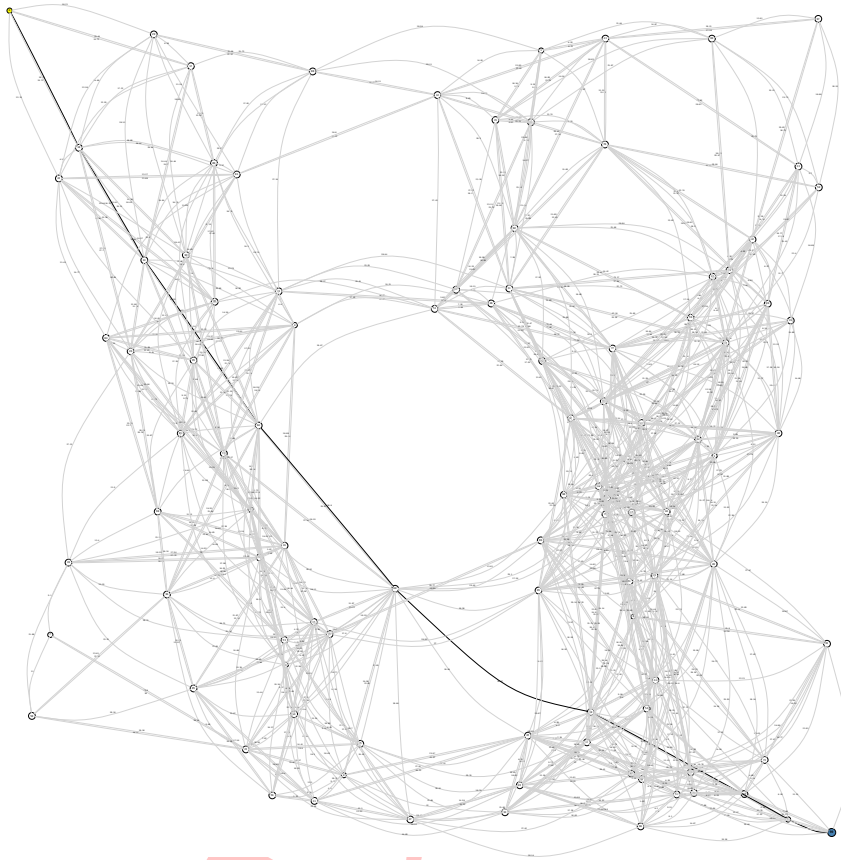
liczbę stanów, może być dokonane tylko poprzez odpowiednio duży eksperyment statystyczny (pomiary dla wielu różnych początkowych plansz sudoku).

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*.

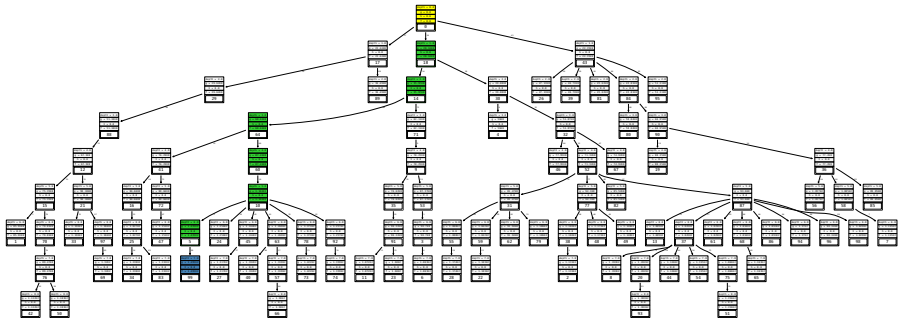


Rys. 2.6: Sztuczny „graf geograficzny" z losowym położeniem 100 wierzchołków w kwadracie jednostkowym (źródło: *opracowanie własne*).

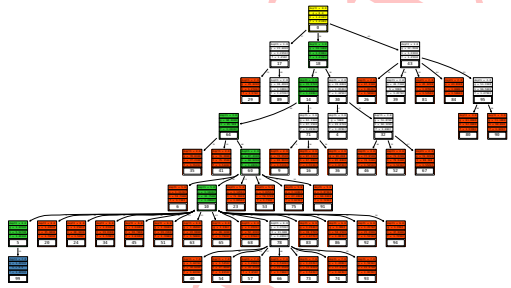
Znając współrzędne geograficzne możemy algorytm A* poinformować tj. wyposażyć go w funkcję heurystyczną obliczającą odległość euklidesową pomiędzy dowolnym stanem a stanem docelowym. Taka informacja znacząco redukuje liczbę odwiedzonych stanów podczas przeszukiwania. Obydwa badane algorytmy znalazły najkrótszą ścieżkę (sekwencję numerów 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].

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

(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

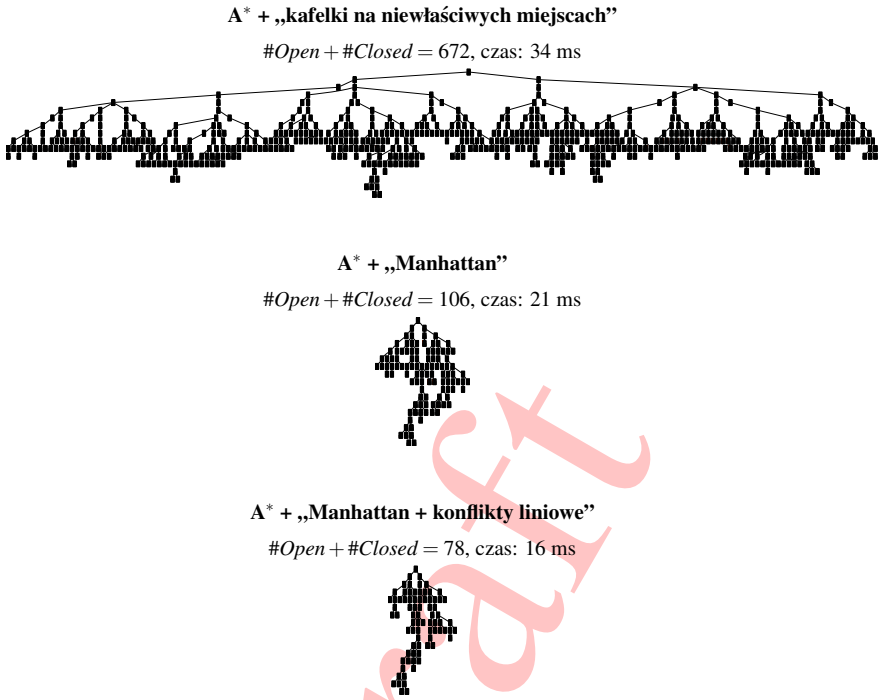


Rys. 2.7: Grafy przeszukiwań wygenerowane przez algorytmy Dijkstry i A* dla grafu z Rys. 2.6 (źródło: *opracowanie własne*).

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 bardziej 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 algorytmu A* mamy pewność, że kolejne heurystyki monotoniczne niosące coraz bardziej dokładną informację o prawdziwym koszcie będą regularnie zmniejszały



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*).

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 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.

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 [44]. 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 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.

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

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 [29] 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

[12] Iteracyjnie pogłębiane A*.

tym samym ustala tzw. *horyzont przeszukiwań* $H = f(s_0) = 0 + h(s_0)$. Następnie, wychodząc od stanu początkowego, algorytm podąża różnymi ściezkami 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 [18] 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.

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: $3.4 \cdot 10^6$, $2.8 \cdot 10^6$	—

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

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')
```

konania (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 (patrz `java.lang.String.hashCode()`); 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 10.2. 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 PuzzleState(PuzzleState 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.

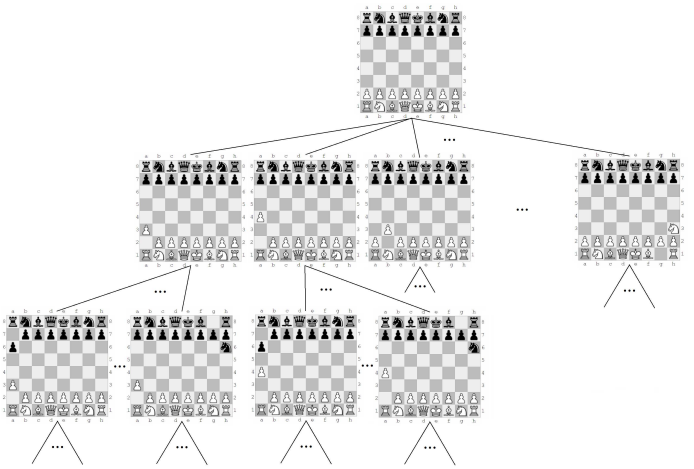
E **Ćwiczenie 2.25** **Zbadaj wpływ zmiany porządku odwiedzania stanów o równej wartości f.** Wskazówka: przygotuj komparator porównujący dwa stany `a` i `b` i zwracający prawdę, gdy $f_a < f_b \vee f_a = f_b \wedge h_a < h_b$; porównaj działanie nowego sposobu porządkowania z poprzednim poprzez odpowiedni 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* [31, 32]. 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 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*)

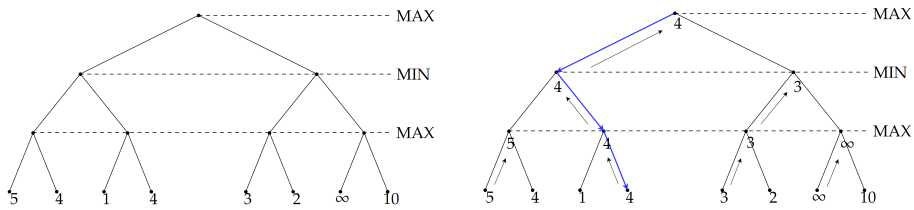
Warto w tym miejscu zwrócić od razu uwagę na dwa istotne elementy. Po pierwsze, w związku z wykładniczym wzrostem rozmiaru drzewa gry wraz z liczbą poziomów (patrz np. rys. 3.1), rzadko kiedy wypłaty obliczane przez algorytmy są dokładne. Najczęściej są to tylko wypłaty przybliżone (lub prawdopodobne) 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 algorytm,

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

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 za-

[2]Na przykład w języku Java: `Double.NEGATIVE_INFINITY`, `Double.POSITIVE_INFINITY`

kończonej 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).
- **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, 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).

lub też $\pm$`Integer.MAX_VALUE`, odpowiednio dla liczb zmiennoprzecinkowych i całkowitych.

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 piony: podwojone, zablokowane, odizolowane; M oznacza mobilność (liczbę dozwolonych ruchów). Symbol $'$ oznacza powyższe wielkości dla strony przeciwnej. 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 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 geometrycznego*. Rolę ilorazu tego ciągu będzie pełnił współczynnik rozgałęziania gry — b. Liczba składników sumy będzie odpowiadała liczbie poziomów drzewa,

[3] Wzajemne wspieranie się.

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

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 on 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 d poziomach, aby poznać dokładną wartość danego stanu gry. 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:

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

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. [5,

11, 33]. Później, w roku 1975, Knuth i Moore oczyścili nieco algorytm i podali szczegółową analizę jego złożoności obliczeniowej w artykule [27]. Pearl dowiódł optymalności tego algorytmu w roku 1982 [37].

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.

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

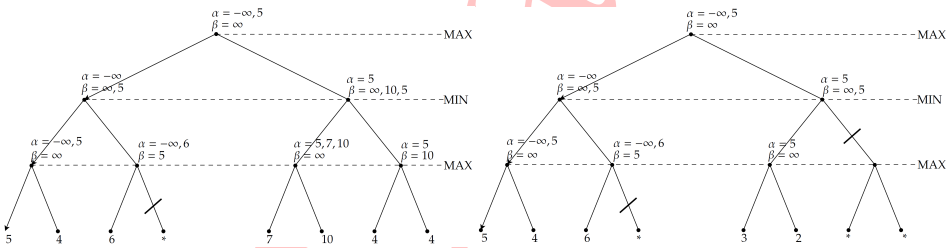
```
1:  procedura AlphaBetaEvaluateMaxState(s, d, D, α, β)
2:      jeżeli IsTerminal(s, d, D) to zwróć h(s)        ▷ h(s) — heurystyczna ocena pozycji
3:      wygeneruj zbiór stanów {t} potomnych dla s
4:      dla wszystkich t wykonaj
5:          v := AlphaBetaEvaluateMinState(t, d + 1/2, D, α, β)
6:          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 kolejnych potomków danego stanu (i ich poddrzewa), ponieważ nie będą one miały

[5] W danym punkcie drzewa.

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 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*)

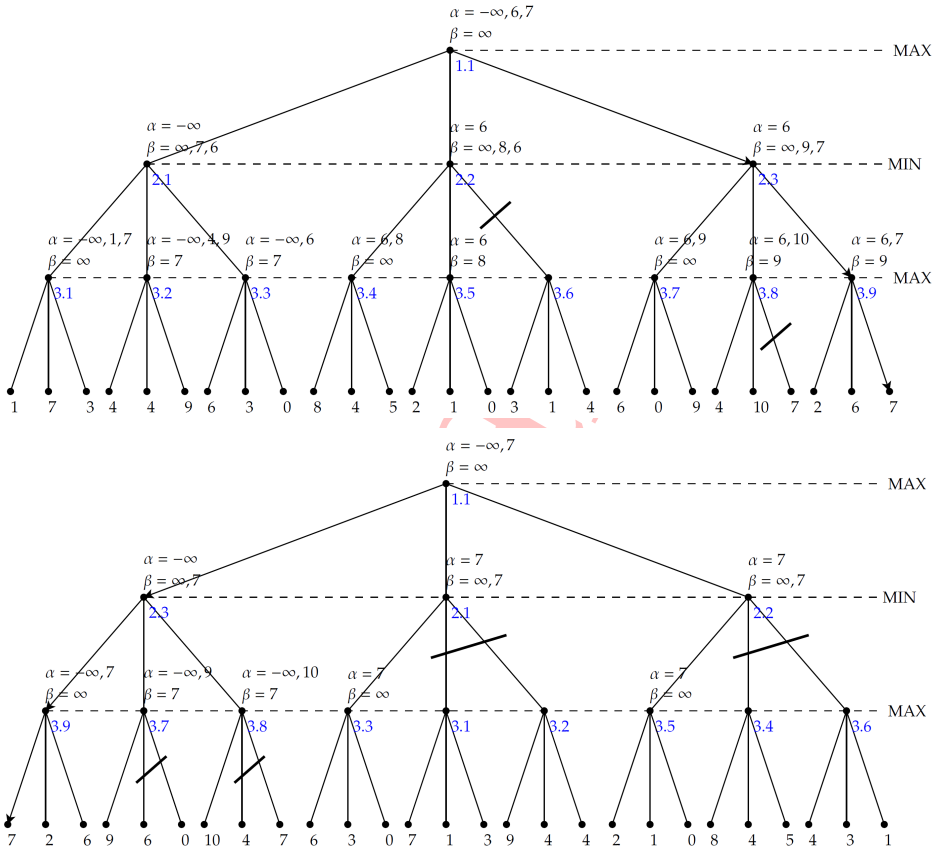
Pomimo redukcji drzewa algorytm „przycinanie α-β” oddaje te same wyniki (oceny ruchów) co algorytm minimax.

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:

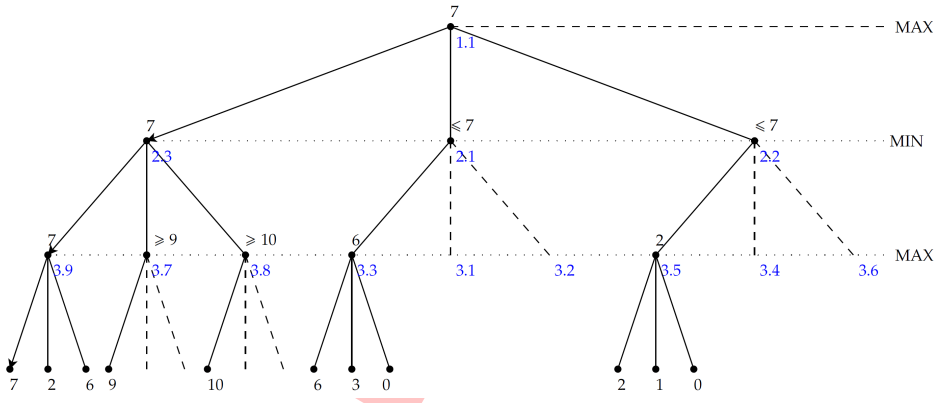
– „przycinania α-β" może nie wykonać żadnego odcięcia i tym samym odwiedzi tyle samo stanów, co algorytm min-max,
– „przycinania α-β" wykona największą możliwą liczbę odcięć (zgodnie ze wzorcem obserwowalnym w dolnej części rys. 3.4), jeżeli najlepszy potomek będzie każdorazowo znajdował się na początku 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 rozgałęziania gry i zadaną maksymalną liczbę poziomów 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. Zdefiniujmy dwie wielkości rekurencyjne:



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*)

- R_d — minimalna liczba stanów (odległych o d poziomów od danego stanu), które trzeba odwiedzić, aby poznać dokładną wartość.
- S_d — minimalna liczba stanów (odległych o d poziomów od danego stanu), które trzeba odwiedzić, aby poznać ograniczenie.

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

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:

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

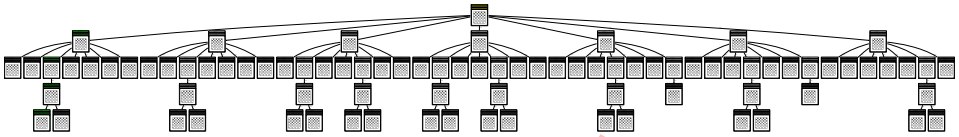
czyli $d/2$-krotne mnożenie współczynnika b. A zatem złożoność $O(b^{d/2})$ wskazuje, że w przypadku optymistycznym algorytm „przycinanie α-β" jest w stanie przeanalizować w tym samym reżimie czasowym dwukrotnie głębsze drzewo niż algorytm min-max, dla którego złożoność obliczeniowa jest klasy $O(b^d)$.

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

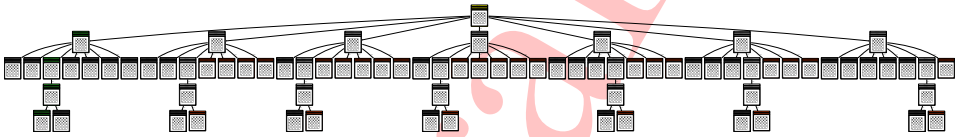
3.2.2 Przykłady działania algorytmów min-max i „przycinanie α-β” dla warcabów

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” wraz z włączonym wariantem Quiescence. Zachęcamy czytelnika do powiększania rysunków.

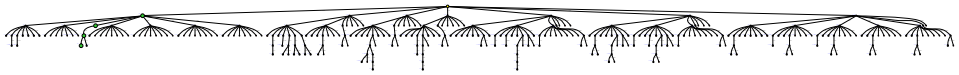


Rys. 3.6: Algorytm *min-max + Quiescence*: zadana głębokość (dla pozycji cichych) 1.0, wygenerowanych stanów 86. (źródło: *opracowanie własne*)



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

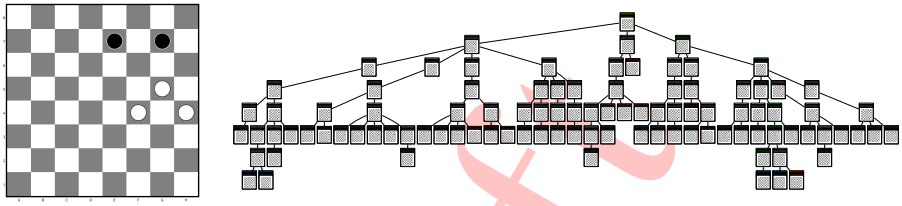
Rys. 3.8: Algorytm *min-max + Quiescence*: zadana głębokość (dla pozycji cichych) 1.5, wygenerowanych stanów 693. (źródło: *opracowanie własne*)



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

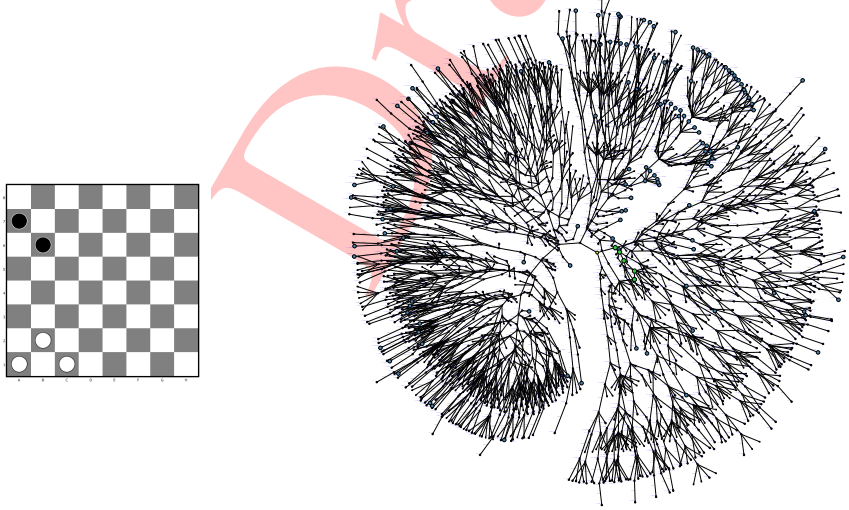
Końcówki warcabowe

Rysunki 3.10–3.13 przedstawiają przykłady końcówek warcabowych przeanalizowanych algorytmem „przycinanie α-β” (z włączonym wariantem Quiescence). Na każdym rysunku kolorem niebieskim oznaczono pozycje zwycięckie. 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.

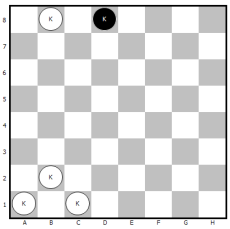


Wariant główny: $(G5:H6, G7:F6, F4:G5, F6:E5, G5:F6, E5:G7, H6:F8:D6)$.

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*)

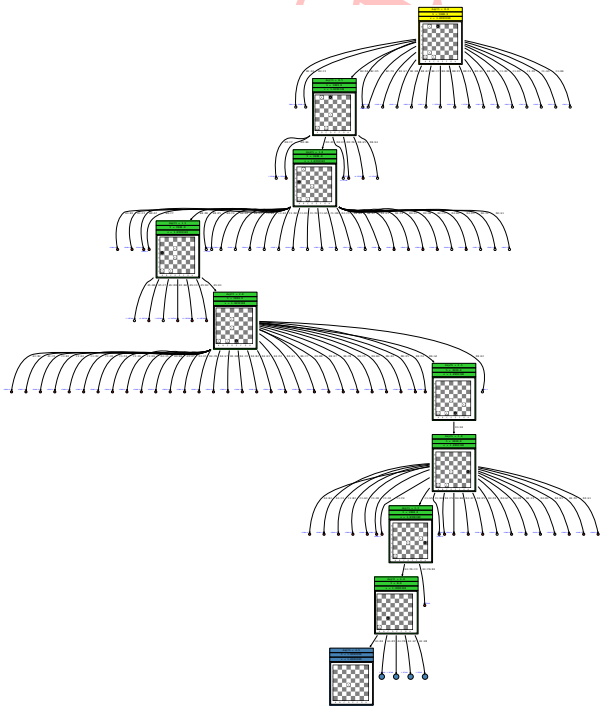


Rys. 3.11: Końcówka warcabowa: kto wygra? Algorytm *„przycinanie α-β” + Quiescence*, zadana głębokość 5.5, wygenerowanych stanów 2845. (źródło: *opracowanie własne*)



```
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” z rys. 3.12. (źródło: *opracowanie własne*)

3.3 Ćwiczenia laboratoryjne (Java + biblioteka *SaC*)

E **Ćwiczenie 3.1** **Napisz grę Connect4 (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 $\times$ 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 Connect4 — 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, bliskość do sufitu, 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 zwycięzkich; 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 z algorytmem przeszukującym, 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`; w imieniu sztucznej inteligencji wykonuj ruch o najlepszej ocenie — ze zbioru ocen ruchów wychwytuje go gotowa metoda `getFirstBestMove()`; do zczytywania 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`.

E **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 Connect4 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 Connect4. 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 Connect4 (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 10.3.

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 Connect4 (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[6], $-\infty$ 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%).

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

II Optymalizacja

4. Metody stochastyczne

TODO

4.1 Algorytm genetyczny

Algorytm genetyczny jest to rodzaj heurystyki przeszukującej przestrzeń alternatywnych rozwiązań problemu w celu wyszukania rozwiązań najlepszych [15, 16, 45]. Metoda została zaproponowana przez Johna H. Hollanda w roku 1975 [23]. Duży wkład w jej rozwój miał David E. Goldberg, który w 1989 opublikował książkę pt. „Genetic Algorithms in Search, Optimization and Machine Learning" [14]. 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 minimum informacji o zadaniu w postaci funkcji celu,

– stosują probabilistyczne metody wyboru.

Algorytmy genetyczne (tak jak wszystkie algorytmy ewolucyjne) korzystają z określeń zapożyczonych z genetyki. Podstawowe określenia są następujące:

– **Populacja** to zbiór osobników o określonej liczebności.
– **Osobnik** to zakodowane rozwiązanie w postaci chromosomów. 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 wymiennie.
– **Chromosom** jest to uporządkowany ciąg genów.
– **Gen** jest podstawową jednostką informacji w chromosomie. Stanowi on pojedynczy element genotypu.
– **Genotyp** jest zespołem chromosomów danego osobnika. Stanowi on podstawę do utworzenia fenotypu.
– **Fenotyp** jest zestawem cech zakodowanym przez dany genotyp. Jest to zbiór cech podlegających ocenie przez funkcje przystosowania.
– **Allel** to wartość danego genu.
– **Locus**[1] jest pozycją wskazującą położenie danego genu w chromosomie.
– **Funkcja przystosowania** (ang. *fitness function*) jest oprócz powyższych określeń bardzo ważnym pojęciem w algorytmach genetycznych. Służy ona do oceny, jak dobrze dany osobnik w populacji jest przystosowany. Funkcja przystosowania jest zdefiniowana przez problem, który należy rozwiązać. Innymi słowy genotyp opisuje proponowane rozwiązanie problemu, a funkcja ocenia, jak dobre jest to rozwiązanie. Funkcja przystosowania pozwala na wybrania 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.
– **Generacją** nazywamy populację osobników w każdej iteracji algorytmu, a nowo utworzoną populację nazywamy **pokoleniem potomków.**

Pseudokod opisany w algorytmie 10 prezentuje klasyczny algorytm genetyczny opisany w szczegółach poniżej. Należy pamiętać, że algorytm genetyczny nie przetwarza pojedynczego rozwiązania, ale cały zbiór nazywany populacją. W wyniku działania algorytmu otrzymujemy rozwiązanie „najlepsze" (czasami jest to zbiór rozwiązań). Rozwiązanie jest najlepsze w sensie probabilistycznym i w ramach zadanych operacji genetycznych i nastaw ilościowych.

[1] Liczba mnoga to *loci*.

Algorytm 10 Algorytm genetyczny

```
1: procedura ALGORYTMGENETYCZNY(T)                    ▷ max. liczba generacji: T
2:     wygeneruj populację początkową
3:     dla t = 1,...,T wykonaj                        ▷ sprawdzenie warunków zatrzymania
4:         ocena przystosowania chromosomów w populacji   ▷ funkcja przystosowania
5:         selekcja chromosomów                           ▷ funkcja selekcji
6:         zastosowanie operatorów genetycznych           ▷ krzyżowanie i mutacja
7:         utworzenie nowej populacji
8:     zwróć „najlepszy" chromosom                    ▷ wyprowadznie „najlepszego" rozwiązania
```

4.1.1 Wybór populacji początkowej

Polega na losowym lub deterministycznym wyborze danej liczby osobników o określonej długości. W klasycznej wersji algorytmu rozmiar populacji jest stały. Niezmienna jest również długość chromosomów, która jest ściśle związana z problemem, dla którego została wygenerowana. Współcześnie stosowane są także inne warianty kształtu populacji i dopuszczają one miedzy 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 rzeczywistymi, 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).

4.1.2 Sprawdzenie warunków zatrzymania

Podstawowym warunkiem zatrzymania się klasycznego algorytmu genetycznego jest osiągnięcie maksymalnej liczby generacji (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 zadanej z góry wartości funkcji przystosowania.

Ostatni warunek ma zastosowanie w zadaniach optymalizacji, gdy znana jest maksymalna (lub minimalna) wartość funkcji przystosowania.

4.1.3 Ocena przystosowania chromosomów w populacji

Funkcja przystosowania ocenia, na ile dobrze dany osobnik jest przystosowany (z punktu widzenia rozwiązywanego problemu). Jest ona ściśle powiązana z problemem, dla którego została zaprojektowana i musi zostać obliczona dla każdego rozwiązania w populacji (dla każdego chromosomu).

4.1.4 Selekcja chromosomów

Metoda selekcji ma za zadanie wybranie najlepiej przystosowanych osobników przy jednoczesnym zachowaniu różnorodności genetycznej populacji. Chromosomy, które zostaną wybrane, będą brały udział w tworzeniu nowej populacji. Wybór zawsze odbywa się zgodnie z zasadą naturalnej selekcji, czyli największe szanse na udział w tworzeniu nowych osobników mają chromosomy o największej wartości funkcji przystosowania. Ze względu na ograniczenie, że rozmiar populacji jest zawsze stały, w wyniku selekcji dany osobnik może zostać wybrany więcej niż jeden raz.

Selekcja koła ruletki

Polega na budowaniu wirtualnego koła, którego wycinki odpowiadają poszczególnym osobnikom. Rozmiar wycinków zależy od wartości funkcji oceny. Każdemu osobnikowi przydzielany jest obszar koła ruletki proporcjonalny do wartości funkcji przystosowania danego chromosomu. Tym samym im lepszy osobnik, tym większy wycinek koła zajmuje, a tym samym ma większe prawdopodobieństwo na zostanie wylosowanym. Zajętość koła dla chromosomu x_i w populacji złożonej z m osobników definiujemy z następującego wzoru:

$$p(x_i) = \frac{f(x_i)}{\sum_{j=1}^{m} f(x_j)}. \tag{4.1}$$

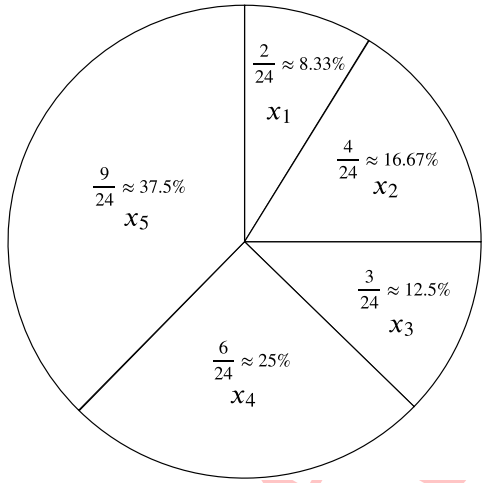
Dla przykładu, mając 5 chromosomów o następujących wartościach funkcji przystosowania:

$$f(x_1) = 2, \quad f(x_2) = 4, \quad f(x_3) = 3, \quad f(x_4) = 6, \quad f(x_5) = 9,$$

otrzymujemy następujące prawdopodobieństwo wylosowania poszczególnych chromosomów:

$$p(x_1) = \frac{2}{24}, \quad p(x_2) = \frac{4}{24}, \quad p(x_3) = \frac{3}{24}, \quad p(x_4) = \frac{6}{24}, \quad p(x_5) = \frac{9}{24}.$$

Selekcja chromosomu, może zostać zwizualizowana jako obrót kołem ruletki. Koło ruletki zbudowane dla powyższego przykładu zostało pokazane na Rys. 4.1.



Rys. 4.1: Koło ruletki — przykładowy podział (źródło: *opracowanie własne*).

Selekcja turniejowa (turniej o rozmiarze k)

Polega na wylosowaniu bez powtórzeń grupy k-elementowej z populacji, a następnie wyborze osobnika o najlepszym przystosowaniu. Całą operację powtarzamy tyle razy, ile jest osobników w populacji. Podczas wyboru najlepszego osobnika możliwe są dwie strategie: wybór deterministyczny lub losowy. Przy wyborze deterministycznym wyboru zawsze dokonujemy z prawdopodobieństwem równym 1 (zawsze wybieramy osobnika najlepiej przystosowanego), w drugim przypadku z prawdopodobieństwem mniejszym od 1. W metodzie można zmieniać wielkość k w trakcie działania algorytmu.

Selekcja rankingowa

Zwana inaczej selekcją rangową. Osobniki ustawiane są kolejno zgodnie z wartością funkcji przystosowania. Rangi zwykle ustala się jako kolejne liczby naturalne, czyli $\{1,2,\ldots,m\}$, gdzie m reprezentuje rangę najlepszego osobnika (o największej wartości funkcji przystosowania). Każdemu osobnikowi przydzielane jest prawdopodobieństwo wylosowania proporcjonalne do jego rangi, które definiujemy według następującego wzoru:

$$p(x_i) = \frac{rank(x_i)}{\sum_{j=1}^{m} rank(x_j)}. \tag{4.2}$$

Sumę rang z mianownika można obliczyć jako sumę ciągu arytmetycznego równą: $m(m+1)/2$.

4.1.5 Zastosowanie operatorów genetycznych

Operatory genetyczne mają na celu rekombinację genów w chromosomach. Wyróżniamy dwa operatory genetyczne: krzyżowanie i mutację. Każdy z operatorów genetycznych wykonywany jest z pewnym prawdopodobieństwem, które definiujemy na początku programu, i które jest stałe w trakcie jego trwania. Typowe zakresy prawdopodobieństw dla poszczególnych operatorów genetycznych są następujące:

- 0.5–1.0 dla krzyżowania,
- 0.0–0.1 dla mutacji.

W 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 jest operacją mającą na celu wymianę materiału genetycznego między osobnikami. W procesie krzyżowania dzielimy całą populację na pary. Następnie dla każdej z par dokonujemy krzyżowania z zadeklarowanym wcześniej prawdopodobieństwem. W przypadku gdy nie jest stosowany operator krzyżowania, wartości genów rodziców są bezpośrednio kopiowane do potomków. Poniżej zostanie przybliżonych kilka wybranych operatorów krzyżowania.

Krzyżowanie jednopunktowe (ang. *1-point crossover*) (1-PX)

Krzyżowanie to polega na wylosowaniu jednego punktu krzyżowania, a następnie wymianie materiału genetycznego, która dokonuje się w następujący sposób:

Pokolenie rodziców:		Pokolenie potomków:
$x_1 = (\mathbf{00110}\vert\mathbf{011})$	$\xrightarrow{\text{krzyżowanie}}$	$(\mathbf{00110}\vert 101)$
$x_2 = (01101\vert 101)$		$(01101\vert\mathbf{011})$

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):

Pokolenie rodziców:		Pokolenie potomków:
$x_1 = (\mathbf{001}\|\mathbf{10}\|\mathbf{011})$	$\xrightarrow{\text{krzyżowanie}}$	$(011\|\mathbf{10}\|101)$
$x_2 = (011\|01\|101)$		$(\mathbf{001}\|01\|\mathbf{011})$

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)

Proces krzyżowania rozpoczyna się od wylosowania dwóch punktów krzyżowania i przekopiowania części genów [13]:

$$\begin{array}{l} x_1 = (\mathbf{58}|\mathbf{213}|\mathbf{764}) \\ x_2 = (78|462|531) \end{array} \xrightarrow{\text{krzyżowanie}} (- - |\mathbf{213}| - - -)$$

W analogicznym segmencie w x_2 znajdują się elementy, które nie zostały skopiowane. Należy je umieścić w potomku. Po wylosowaniu dwóch punktów krzyżowania posiadamy następujące odwzorowania: $4 \to 2, 6 \to 1, 2 \to 3$. Dla przykładu $6 \to 1$ oznacza, że należy skopiować 6 w miejsce 1:

$$\begin{array}{l} x_1 = (\mathbf{58}|\mathbf{213}|\mathbf{764}) \\ x_2 = (78|462|531) \end{array} \xrightarrow[6\to1]{\text{krzyżowanie}} (- - |\mathbf{213}| - - 6)$$

Podobnej operacji nie możemy wykonać dla $4 \to 2$ gdyż **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 3:

$$\begin{array}{l} x_1 = (\mathbf{58}|\mathbf{213}|\mathbf{764}) \\ x_2 = (78|462|531) \end{array} \xrightarrow[4\to2\to3]{\text{krzyżowanie}} (- - |\mathbf{213}| - 46)$$

Pozostałe geny kopiujemy z x_2:

$$\begin{array}{l} x_1 = (\mathbf{58}|\mathbf{213}|\mathbf{764}) \\ x_2 = (78|462|531) \end{array} \xrightarrow{\text{krzyżowanie}} (78|\mathbf{213}|546)$$

Drugiego potomka generujemy zamieniając x_1 i x_2 miejscami:

Pokolenie rodziców:		Pokolenie potomków:
$x_1 = (\mathbf{58}\vert\mathbf{213}\vert\mathbf{764})$	$\xrightarrow{\text{krzyżowanie}}$	$(78\vert\mathbf{213}\vert546)$
$x_2 = (78\vert462\vert531)$		$(\mathbf{58}\vert462\vert\mathbf{713})$

Krzyżowanie z zachowaniem porządku (ang. *order crossover*) (OX)

Podobnie jak w przypadku poprzednim losowane są dwa punktu w chromosomach rodziców [9]. 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 mamy:

$x_1 = (\mathbf{58}\vert\mathbf{213}\vert\mathbf{764})$	$\xrightarrow{\text{krzyżowanie}}$	$(\ -\ -\ \vert\mathbf{213}\vert\ -\ -\ -\ )$
$x_2 = (78\vert462\vert531)$		$(\ -\ -\ \vert462\vert\ -\ -\ -\ )$

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:

Pokolenie rodziców:		Pokolenie potomków:
$x_1 = (\mathbf{58}\vert\mathbf{213}\vert\mathbf{764})$	$\xrightarrow{\text{krzyżowanie}}$	$(46\vert\mathbf{213}\vert578)$
$x_2 = (78\vert462\vert531)$		$(\mathbf{13}\vert462\vert\mathbf{758})$

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 [35]. Na początek losujemy gen i rodzica, od którego zaczynamy krzyżowanie. Powiedzmy, że zaczniemy od **5** w pierwszym rodzicu x_1:

$x_1 = (\mathbf{58213764})$	$\xrightarrow{\text{krzyżowanie}}$	$(\mathbf{5}\ -\ -\ -\ -\ -\ -\ -\ )$
$x_2 = (78462315)$		

5 leży na tym samym miejscu co 7, więc do chromosomu dodajmy **7** w miejscu z pierwszego chromosomu:

$$\begin{array}{l} x_1 = (\mathbf{58213764}) \\ x_2 = (78462315) \end{array} \xrightarrow{\text{krzyżowanie}} (\mathbf{5}\text{ - - - - }\mathbf{7}\text{ - - })$$

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:
$x_1 = (\mathbf{58213764})$	$\xrightarrow{\text{krzyżowanie}}$	$(\mathbf{58263714})$
$x_2 = (78462315)$		$(\mathbf{78}4\mathbf{1}23\mathbf{6}5)$

Operator mutacji

Mutacja ma za zadanie wprowadzić różnorodność genetyczną populacji. Dla chromosomów kodujących informację binarnie mutacja polega na zamianie wartości bitu na przeciwny:

$$(00110010101) \xrightarrow{\text{mutacja}} (00110011101)$$

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:

$$(123\mathbf{456}789) \xrightarrow{\text{mutacja}} (126\mathbf{543}789)$$

Alternatywnie można wybrać dany zakres genów w chromosomie (lub wybrać nawet pojedyncze geny) i dokonać permutacji znajdujących się w nich wartości:

$$(123\mathbf{456}789) \xrightarrow{\text{mutacja}} (125\mathbf{364}789)$$

4.1.6 Utworzenie nowej populacji

Chromosomy 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.

4.1.7 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 wcale nie muszą być sobie równoważne, ponieważ najlepsze rozwiązanie w całej historii, może zostać utracone w wyniku działania algorytmu. Czasami aby zapobiegać sytuacji tego typu istnieje opcja „elitaryzmu", która w każdej iteracji zastępuje najsłabszy chromosom najlepszym, aby go nie utracić.

4.2 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.

4.2.1 Dyskretny problem plecakowy

Dyskretny problem plecakowy często przedstawia się jako problem złodzieja rabującego sklep — znalazł on n 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 plecaka C. Przedmiotów nie można dzielić (dzielenie byłoby możliwe w ciągłym problemie plecakowym).

Formalna definicja problemu jest następująca. Do dyspozycji posiadamy n przedmiotów, a każdy z nich jest opisany jako para (v_i, c_i):

$$A = \{(v_i, c_i)\}_{i=1\ldots n}. \tag{4.3}$$

Zadaniem jest znalezienie takiego podzbioru B, w którym wartość przedmiotów jest maksymalna, ale ich objętość nie przekracza maksymalnej objętości plecaka, czyli:

$$B \subseteq A, \qquad \sum_{(v_i,c_i)\in B} v_i \longrightarrow \max, \qquad \sum_{(v_i,c_i)\in B} c_i \leqslant C. \tag{4.4}$$

Na potrzeby niniejszego zadania użyjemy chromosomów binarnych, w których każdy z genów może przyjmować tylko jedną wartość 0 lub 1.

Każdy chromosom będzie reprezentowała jedno możliwe rozwiązanie tj. pewien podzbiór przedmiotów, które mogą 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 poniżej. Chromosom koduje informacje, o tym że w plecaku znajdują się przedmioty: $1, 2, 6, 8, 14$.

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

Funkcja przystosowania dla dyskretnego problemu plecakowego będzie sumą wartości przedmiotów, jeżeli ich sumaryczna objętości nie przekracza maksymalnej objętości plecaka C, w przeciwnym razie będzie to 0. Matematycznie funkcję przystosowania dla danego osobnika można zapisać jako:

$$p(B) = \begin{cases} \sum_{(v_i,c_i)\in B} v_i, & \sum_{(v_i,c_i)\in B} c_i \leqslant C; \\ 0, & \sum_{(v_i,c_i)\in B} c_i > C. \end{cases} \tag{4.5}$$

Podczas losowania populacji początkowej należy pamiętać, żeby wszystkie chromosomy w pierwszej generacji nie zostały ocenione przez funkcje przystosowania opisaną wzorem (4.5) na 0. Taka sytuacja może mieć miejsce, kiedy geny

w populacji początkowe losujemy np. z prawdopodobieństwem 0.5. Powoduje to, że w każdym plecaku znajduje się około połowa wszystkich przedmiotów. W połączeniu z relatywnie małą wartość C ($C \ll \sum_{i=1}^{n} c_i$), funkcja przystosowania oceni każdy plecak jako przepełniony. Rozwiązaniem tego problemu może być zmniejszenie prawdopodobieństwa pojawienia się przedmiotu w plecaku w populacji początkowej. Ewentualnie należy zmodyfikować wzór (4.5), tak żeby algorytm w pierwszej kolejności starał się znaleźć taki podzbiór przedmiotów, który będzie mieścił się w plecaku:

$$p(B) = \begin{cases} \sum_{(v_i,c_i)\in B} v_i, & \sum_{(v_i,c_i)\in B} c_i \leqslant C; \\ \frac{1}{\sum_{(v_i,c_i)\in B} v_i}, & \sum_{(v_i,c_i)\in B} c_i > C. \end{cases} \tag{4.6}$$

Rozwiązanie dokładne

Dyskretny problem plecakowy może zostać rozwiązany przy pomocy programowania dynamicznego. Korzystając z indukcji staramy się odnaleźć $V_{i,j}$ — wartość najlepszego upakowania plecaka o objętości j, za pomocą przedmiotów o numerach $1,\ldots,i$. Wartość tą definiujemy następująco przy użyciu rekurencji:

$$\begin{aligned} &V_{0,j} = 0, \\ &V_{i,0} = 0, \\ &V_{i,j} = V_{i-1,j}, && \text{jeżeli} \quad c_i > j, \\ &V_{i,j} = \max(V_{i-1,j},\, V_{i-1,j-c_i} + v_i), && \text{jeżeli} \quad c_i \leqslant j. \end{aligned} \tag{4.7}$$

Przekładając powyższy wzór na pseudokod otrzymujemy Algorytm 11. Podany algorytm podaje tylko wartość najlepszego upakowania, ale nie definiuje jaki jest to podzbiór przedmiotów. W celu znalezienia podzbioru przedmiotów należałoby wprowadzić drugą tablicę tzw. wskazań wstecznych (ang. *back-pointers*) i na końcu ją prześledzić. Tego typu rozwiązanie stosuje się w wielu podejściach opartych na programowaniu dynamicznym. Złożoność obliczeniowa zaprezentowanego algorytmu wynosi $\Theta(n\,C)$. Złożoność ta pozornie wydaje się liniowa i może dojść do sytuacji kiedy C będzie skalowało się np. proporcjonalnie do 2^n, co powoduje że problem może wymagać czasu wykładniczego (dlatego zadanie jest problemem NP-trudnym).

4.2.2 Problem komiwojażera

Drugim klasycznym problemem na, przykładzie którego bardzo często tłumaczy się działanie algorytmu genetycznego, jest problem komiwojażera (ang. *travelling salesman problem*) (TSP). Problem ten definiuje się jako problem komiwojażera, który ma odwiedzić n miast i chce pokonać jak najkrótszą odległość. Jest to problem NP-trudny. W implementacji rozwiązania tego problemu należy posłużyć się

Algorytm 11 Algorytm rozwiązywania dyskretnego problemu plecakowego

```
1:  procedura AlgorytmDyskretnyProblemPlecakowy
2:      dla i = 0,...,n wykonaj
3:          V_{i,0} := 0
4:      dla j = 0,...,C wykonaj
5:          V_{0,j} := 0
6:      dla i = 0,...,n wykonaj                ▷ bierzemy pod uwagę i pierwszych przedmiotów
7:          dla j = 0,...,C wykonaj
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}
```

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. Wartość każdego genu będzie stanowić numer miasta. Przykład takiego chromosomu znajduje się na diagramie poniżej.

5	2	14	8	4	3	10	15	6	7	13	9	12	1

Zaprezentowany chromosom definiuje kolejność, w jakiej komiwojażer odwiedzałby miasta. Wartość funkcji przystosowania można w prosty sposób zdefiniować jako sumę odległości poszczególnych połączeń pomiędzy miastami, które musi pokonać komiwojażer:

$$f(x_i) = \|x_{i,n}\, x_{i,1}\| + \sum_{j=1}^{n-1} \|x_{i,j}\, x_{i,(j+1)}\|, \tag{4.8}$$

gdzie przez $x_{i,j}$ rozumiemy j-ty gen w i-tym chromosomie, $\|\cdot\|$ jest to odległość pomiędzy dwoma miastami. Jednakże problem postawiony w ten sposób przybiera postać zadania minimalizacji. Teoretycznie w klasycznym algorytmie genetycznym zawsze powinno sprowadzić się zadanie do zadania maksymalizacji. Przykład funkcji selekcji spełniającej to kryterium w omawianym problemie jest następująca:

$$f(x_i) = 1 - \frac{\|x_{i,n}\, x_{i,1}\| + \sum_{j=1}^{n-1} \|x_{i,j}\, x_{i,(j+1)}\|}{f_{\max} - f_{\min}}, \tag{4.9}$$

gdzie $f_{\min}$ to najkrótsza, a $f_{\max}$ to najdłuższa możliwa odległość łącząca wszystkie miasta. W praktyce nie znamy wartości stałych z mianownika, ale $f_{\min}$ możemy

przyjąć jako 0 i pominąć. $f_{\max}$ możemy przyjąć jako najdłuższą odległość łączącą miasta spośród wszystkich odległości kodowanych przez chromosomy z populacji początkowej. Należy zauważyć, że część funkcji selekcji może zostać użyta również w zadaniach minimalizacji — taką funkcją jest m.in. selekcja turniejowa. W przypadku problemu komiwojażera należy stosować odpowiednie operatory genetyczne, takie jak np. krzyżowania z częściowym odwzorowaniem i mutację przez inwersję.

4.3 Ćwiczenia laboratoryjne (MATLAB)

E **Ćwiczenie 4.1 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.

E **Ćwiczenie 4.2 Napisz funkcje realizujące selekcje rankingową i turniejową oraz krzyżowanie dwupunktowe (wykorzystaj program z Ćwiczenia 4.1).** 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.

E **Ćwiczenie 4.3 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 4.1 dla wylosowanego problemu plecakowego (lub kilku) sprawdź, czy algorytm genetyczny zwraca ten sam wynik, co rozwiązanie dokładne.

E **Ćwiczenie 4.4 Napisz skrypt rozwiązujący problem komiwojażera za pomocą algorytmu genetycznego (wykorzystaj program z Ćwiczenia 4.1 oraz 4.2).** 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.

III Uczenie maszynowe

5. Perceptrony

5.1 Perceptron prosty

Pomysłodawcą pierwszego układu uczącego się był Frank Rosenblatt, który w artykule [42] 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 metodą prób i błędów, który realizują ludzie na podstawie obserwowanych przykładów. Rosenblatt był psychologiem i neurobiologiem, a formalny dowód zbieżności dla zaproponowanego przezeń algorytmu dostarczył w 1962 r. Novikoff [34]. Odkrycia te zapoczątkowały dalszy rozwój dziedzin 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”

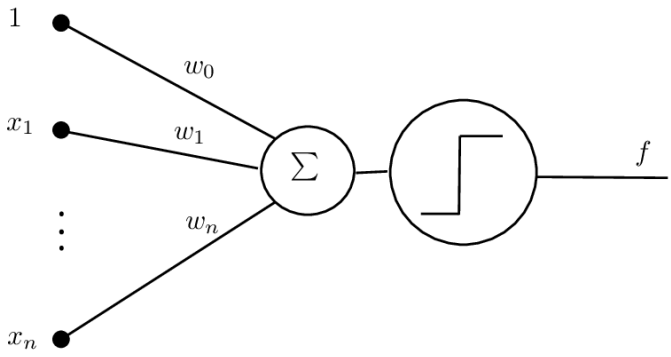
ona w postaci sumy ważonej numeryczne cechy obiektów (tzw. cechy) z pewnymi współczynnikami, które perceptron ma odpowiednio dobrać w procesie uczenia.

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.

5.1.1 Schemat graficzny

Obliczenia w perceptronie prostym są realizowane zgodnie ze schematem przedstawionym na Rys. 5.1 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 neuronu*, a w przypadku perceptronu prostego jest to funkcja schodkowa (zazna-

[1] Lub inaczej: z nauczycielem.



Rys. 5.1: Schemat graficzny perceptronu prostego. (źródło: *opracowanie własne*)

czona 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:

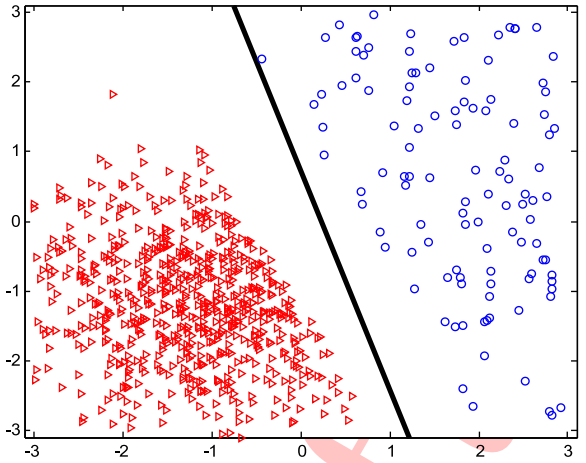
$$s = w_0 + w_1 x_1 + \cdots + w_n x_n. \tag{5.1}$$

$$f(s) = \begin{cases} 1, & \text{dla } s > 0; \\ -1, & \text{dla } s \leqslant 0. \end{cases} \tag{5.2}$$

5.1.2 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. 5.2, 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, a w pokazanej sytuacji można wskazać wiele innych prostych (nieskończenie wiele) prawidłowo separujących dane.

[2]Perceptron prosty wymaga cech rzeczywistoliczbowych. Jest to ograniczenie oznaczające niemożność pracy na zmiennych (cechach) wyliczeniowych, takich jak np. kolor oczu.



Rys. 5.2: Przykład klasyfikacji binarnej na płaszczyźnie. (źródło: *opracowanie własne*)

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:

$$w_0 + w_1x_1 + w_2x_2 + \cdots + w_nx_n = 0. \tag{5.3}$$

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:

$$\forall i, y_i = 1 \quad w_0 + w_1x_{i1} + w_2x_{i2} + \cdots + w_nx_{in} > 0. \tag{5.4}$$

i

$$\forall i, y_i = -1 \quad w_0 + w_1x_{i1} + w_2x_{i2} + \cdots + w_nx_{in} \leqslant 0. \tag{5.5}$$

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

Patrząc ponownie na schemat graficzny (rys. 5.1) 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: $\langle \mathbf{w}, \mathbf{x}_i \rangle$:

$$\langle \mathbf{w}, \mathbf{x}_i \rangle = \sum_{j=0}^{n} w_j x_{ij}. \tag{5.6}$$

5.1.3 Algorytm uczenia on-line dla perceptronu prostego

Podany poniżej algorytm 12, nazywany zwyczajowo „regułą perceptronu”[3], przedstawia sposób uczenia on-line dla perceptronu prostego. Uczenie on-line oznacza

Algorytm 12 „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** $\mathbf{w}(k)$

w ogólności, że poprawki współczynników wagowych odbywają się na podstawie obejrzanego pojedynczego przykładu[4]. Oprócz zbioru danych uczących D dodatkowym argumentem podawanym przez użytkownika na wejście algorytmu jest liczba $\eta \in (0, 1]$ nazywana *współczynnikiem uczenia* (ang. *learning rate* lub *learning*

[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.

coefficient). Współczynnik ten decyduje o wielkości dokonywanych poprawek i stanowi on analogię do długości kroku w metodach optymalizacji. 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 wg wzoru:

$$\mathbf{w}(k+1) := \mathbf{w}(k) + \eta y_i \mathbf{x}_i. \tag{5.7}$$

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 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 $y_i \langle \mathbf{w}(k), \mathbf{x}_i \rangle \leqslant 0$. 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:

$$e_i = -y_i \langle \mathbf{w}(k), \mathbf{x}_i \rangle. \tag{5.8}$$

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.:

$$\mathbf{w}(k+1) := \mathbf{w}(k) - \frac{\partial e_i}{\partial \mathbf{w}(k)}. \tag{5.9}$$

Łatwo sprawdzić, że

$$\frac{\partial e_i}{\partial \mathbf{w}(k)} = -y_i \mathbf{x}_i, \tag{5.10}$$

co uzasadnia postać wzoru (5.7) 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 12, czyli w efekcie iteracyjnego wykonywania wzoru (5.7) na rzecz różnych punktów danych. Teoretycznie możliwe jest pojawienie się oscylacji w poprawkach i brak zbieżności algorytmu. Twierdzenie i jego dowód przedstawione w kolejnym rozdziale zaprzeczą takiej możliwości.

5.1.4 Twierdzenie o zbieżności algorytmu uczącego

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

Definicja 5.1.1 — 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:

$$\forall i, y_i = 1 \quad \langle \mathbf{w}^*, \mathbf{x}_i \rangle > 0, \tag{5.11}$$

$$\forall i, y_i = -1 \quad \langle \mathbf{w}^*, \mathbf{x}_i \rangle \leqslant 0. \tag{5.12}$$

Twierdzenie 5.1.1 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

$$R = \max_{i=1,\ldots,m} \|\mathbf{x}_i\|. \tag{5.13}$$

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

oznacza tzw. *margines separacji* zdefiniowany jako:

$$\gamma = \min_{i=1,\ldots,m} \frac{y_i \langle \mathbf{w}^*, \mathbf{x}_i \rangle}{\|(w_1^*, \ldots, w_n^*)\|}. \tag{5.14}$$

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 (5.7) na poprawkę wag.

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

$$\begin{aligned}
\langle \mathbf{w}^*, \mathbf{w}(k) \rangle &= \langle \mathbf{w}^*, \mathbf{w}(k-1) + \eta y_i \mathbf{x}_i \rangle = \langle \mathbf{w}^*, \mathbf{w}(k-1) \rangle + \eta \underbrace{y_i \langle \mathbf{w}^*, \mathbf{x}_i \rangle}_{\geqslant \gamma} \\
&\geqslant \langle \mathbf{w}^*, \mathbf{w}(k-1) \rangle + \eta\gamma \\
&\geqslant \langle \mathbf{w}^*, \mathbf{w}(k-2) \rangle + 2\eta\gamma \\
&\vdots \\
&\geqslant \underbrace{\langle \mathbf{w}^*, \mathbf{w}(0) \rangle}_{0} + k\eta\gamma.
\end{aligned} \tag{5.15}$$

Nierówność $\langle \mathbf{w}^*, \mathbf{w}(k) \rangle \geqslant k\eta\gamma$ oznacza, że obserowany iloczny skalarny rośnie w trakcie 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):

$$\begin{aligned}\|\mathbf{w}(k)\|^2 = \langle \mathbf{w}(k), \mathbf{w}(k)\rangle &= \langle \mathbf{w}(k-1) + \eta y_i \mathbf{x}_i, \mathbf{w}(k-1) + \eta y_i \mathbf{x}_i\rangle \\ &= \|\mathbf{w}(k-1)\|^2 + \underbrace{2\eta y_i \langle \mathbf{w}(k-1), \mathbf{x}_i\rangle}_{\leqslant 0} + \eta^2 \underbrace{y_i^2}_{1} \underbrace{\|\mathbf{x}_i\|^2}_{\leqslant R^2} \\ &\leqslant \|\mathbf{w}(k-1)\|^2 + \eta^2 R^2 \\ &\leqslant \|\mathbf{w}(k-2)\|^2 + 2\eta^2 R^2 \\ &\vdots \\ &\leqslant \underbrace{\|\mathbf{w}(0)\|^2}_{0} + k\eta^2 R^2. \end{aligned} \tag{5.16}$$

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

Otrzymane ograniczenia nierównościowe (5.15) i (5.16) łączymy za pomocą nierówności Cauchy'ego-Schwarza — $\langle a, b\rangle \leqslant \|a\|\|b\|$ — otrzymując:

$$k\eta\gamma \leqslant \langle \mathbf{w}(k), \mathbf{w}^*\rangle \leqslant \|\mathbf{w}(k)\| \underbrace{\|\mathbf{w}^*\|}_{1} \leqslant \sqrt{k}\eta R. \tag{5.17}$$

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:

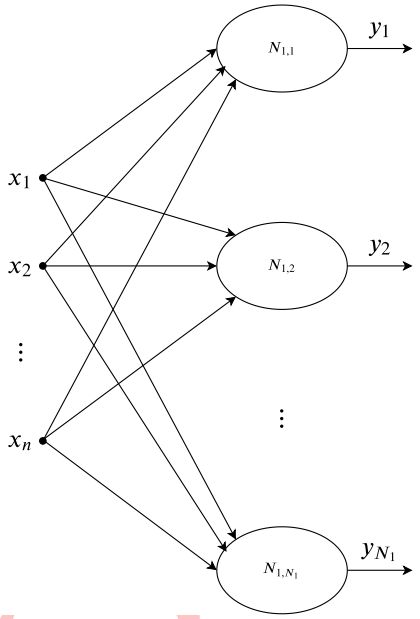
$$\begin{aligned} k\eta\gamma &\leqslant \sqrt{k}\eta R \\ k &\leqslant \frac{R^2}{\gamma^2}. \end{aligned} \tag{5.18}$$

■

5.2 Jednokierunkowe sieci wielowarstwowe

Sztuczne neurony mogą tworzyć różnego rodzaju większe struktury zwane sieciami neuronowymi [30, 45]. Najprostszą strukturą jest tzw. sieć jednowarstwowa. Schemat sieci jednowarstwowej prezentuje rys. 5.3. 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. 5.4.

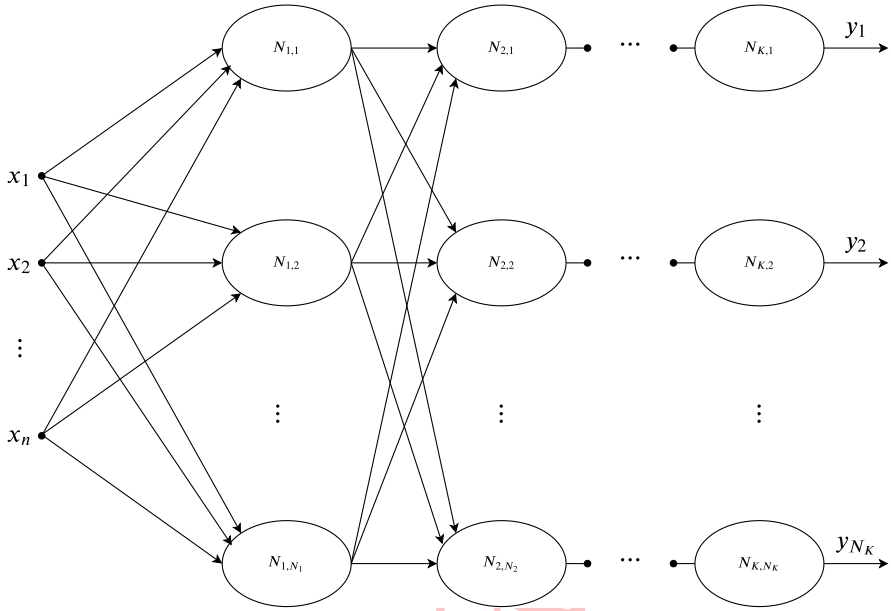


Rys. 5.3: Sieć jednowarstwowa. Użyta ogólna notacja $N_{k,i}$ oznacza i-ty neuron w warstwie k-tej. (źródło: *opracowanie własne*)

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 liniowej za pomocą *funkcji sigmoidalnych*, czyli funkcji postaci

$$f(s) = \frac{1}{1+e^{-s}}, \tag{5.19}$$

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 [28]. Z kolei twierdzenie Cybenki mówi, że sigmoidalna sieć neuronowa ma uniwersalne właściwości aproksymacyjne [8] oraz że dla dowolnej funkcji ciągłej można dobrać odpowiednią sigmoidalną



Rys. 5.4: Sieć wielowarstwowa. (źródło: *opracowanie własne*)

sieć neuronową, która ją przybliży. Niestety, powyższe rezultaty są egzystencjalne, a nie konstruktywne, tzn. stwierdzają istnienie wspomnianych własności, jednak nie podają konkretnego przepisu, który mówiłby, w jaki sposób uzyskać właściwy aproksymator dla podanego zbioru danych.

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.

Zakładając, że rozpatrujemy sieć neuronową posiadającą K warstw, należy zauważyć, że wyjście neuronu i w warstwie k będzie jednocześnie i-tym wejściem dowolnego neuronu j w warstwie $k+1$. Korzystając z tej zależności można zapisać,

[5] Inaczej: generalizacji.

że $x_{k+1,i} = y_{k,i}$, co uogólniając można przedstawić w postaci zapisu:

$$x_{k,i} = \begin{cases} x_i, & \text{dla } k = 1 \\ y_{k+1,i}, & \text{dla } k = 2,\ldots,K \\ 1, & \text{dla } i = 0,\ k = 1,\ldots,K \end{cases}, \tag{5.20}$$

gdzie przez $x_{k,0}$ rozumiemy wejście progowe, na które zawsze podawana jest wartość 1. Wektor wszystkich wag neuronu i w warstwie k może zapisać jako:

$$\mathbf{w}_{k,i} = \left(w_{k,i,0},\ldots,w_{k,i,N_{k-1}}\right), \qquad k = 1,\ldots,K, \quad i = 1,\ldots,N_k, \tag{5.21}$$

gdzie przez N_{k-1} rozumiemy liczbę neuronów w warstwie $k-1$. Sygnał wyjściowy dowolnego neuronu $N_{k,i}$ możemy opisać jako:

$$y_{k,i} = f(s_{k,i}), \tag{5.22}$$

gdzie f jest *funkcją aktywacji neuronu*. Jej argumentem jest suma ważona sygnałów wejściowych (lub inaczej — iloczyn skalarny wektora wag i wejść):

$$s_{k,i} = \sum_{j=0}^{N_{k-1}} w_{k,i,j} \cdot x_{k,j}. \tag{5.23}$$

Przykłady powszechnie stosowanych funkcji aktywacji zostaną podane w kolejnym punkcie.

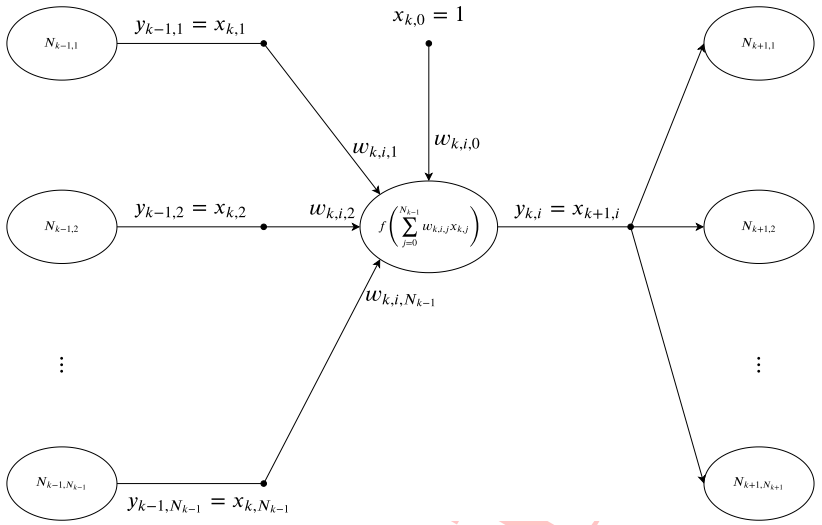
Schemat działania pojedynczego i-tego neuronu w warstwie k przedstawia rys. 5.5. Należy zauważyć, że sygnały wyjściowe neuronów w warstwie K:

$$y_{K,1}, y_{K,2}, \ldots, y_{K,N_K}, \tag{5.24}$$

są jednocześnie sygnałami wyjściowymi dla całej sieci neuronowej [39].

5.2.1 Funkcje aktywacji neuronu

Neurony znajdujące się w wewnętrznych warstwach sieci z reguły są typu sigmoidalnego. Czasami używa się również neuronów o innej funkcji aktywacji. Warstwa wyjściowa często składa się z neuronów, których funkcja aktywacji przyjmuje postać liniową. Najczęściej używane funkcje aktywacji wraz z ich pochodnymi zostały przedstawione w tabeli 5.1. Należy zaznaczyć, że pochodne odgrywają ważną rolę podczas uczenia sieci.

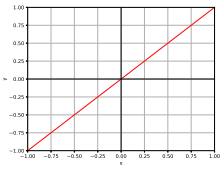
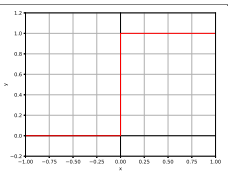
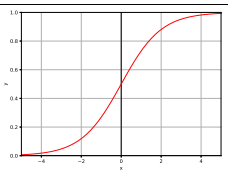
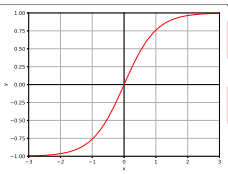
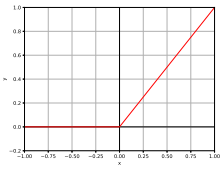


Rys. 5.5: Schemat działania i-tego neuronu w warstwie k. (źródło: *opracowanie własne*)

5.3 Algorytm wstecznej propagacji błędów

Podstawową metodą uczenia jednokierunkowej sieci neuronowej jest ***algorytm wstecznej propagacji błędów*** (ang. *backpropagation*) [30, 45] opracowany w 1986 r. przez Davida Rumelharta, Geoffrey'a Hintona i Ronalda Williamsa [43]. 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. Powszechnie do korekcji parametrów (wag) sieci używana jest metoda najmniejszych kwadratów, tzn. wspomniany błąd pomiędzy odpowiedzią sieci a wyjściem oczekiwanym 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 bliżej 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.

Tabela 5.1: 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		$f(s) = \begin{cases} 0, & \text{dla } s < 0; \\ 1, & \text{dla } s \geqslant 0. \end{cases}$	$f'(s) = 0$ 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		$f(s) = \begin{cases} 0, & \text{dla } s \leqslant 0; \\ s & \text{dla } s > 0. \end{cases}$	$f'(s) = \begin{cases} 0, & \text{dla } s \leqslant 0; \\ 1, & \text{dla } s > 0. \end{cases}$

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

$$\frac{1}{2}e^2 = \frac{1}{2}\sum_{i=1}^{N_K}(y_i - y_{K,i})^2. \tag{5.25}$$

Ogólny wzór na gradientową korektę pewnej wagi w sieci możemy zapisać następująco:

$$w_{k,i,j}(t+1) = w_{k,i,j}(t) - \eta \frac{\partial \frac{1}{2}e^2}{\partial w_{k,i,j}}. \tag{5.26}$$

Indeks t zapisany w nawiasie oznacza numer kroku w algorytmie. Po wykonaniu odpowiednich przekształceń, algorytm wstecznej propagacji błędu określa następujący wzór na poprawioną wartość dowolnej wagi w sieci:

$$w_{k,i,j}(t+1) = w_{k,i,j}(t) + 2\eta\ \delta_{k,i}\ x_{k,j}, \tag{5.27}$$

$$\delta_{k,i} = \varepsilon_{k,i}\ f'(s_{k,i}), \tag{5.28}$$

gdzie jako $\varepsilon_{k,i}$ rozumiemy błąd i-tego neuronu w warstwie k:

$$\varepsilon_{k,i} = \begin{cases} y_i - y_{K,i}, & k = K; \\ \sum_{j=1}^{N_{k+1}} \delta_{k+1,j}\ w_{k+1,i,j}, & k = 1,\ldots,K-1. \end{cases} \tag{5.29}$$

Istotnym jest fakt, że wyrażenia $\varepsilon_{k,i}$ oraz $\delta_{k,i}$ związane z propagowanymi błędami są w algorytmie obliczane od prawej strony schematu sieci ku lewej, tj. zgodnie z malejącą kolejnością indeksu $k = K, K-1, \ldots, 1$.

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 m wszystkich składników). Mówi się wówczas o tzw. gradiencie stochastycznym (lub losowym), co oznacza, że proces uczenia przypomina losowe błądzenie, ponieważ możemy mieć do czynienia z oscylacjami w trajektorii wektora wag.

Bardzo ważnym problemem poruszanym w tematyce sieci neuronowych jest początkowa inicjalizacja wag. Im wagi będą bliżej wartości optymalnych, tym uczenie sieci będzie trwało krócej. Dodatkowo w przypadku sigmoidalnej funkcji aktywacji 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.

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 [39].

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 (ang. *resilient backpropagation*) [41], metoda gradientów sprzężonych, algorytm Levenberga-Marquardta [17] czy uczenie oparte na rekurencyjnej metodzie najmniejszych kwadratów [3].

5.3.1 Przykład prostej sieci neuronowej

Sieci neuronowe mogą tworzyć różne skomplikowane struktury. W tym miejscu postaramy się przybliżyć jedną z podstawowych i najprostszych architektur sieci neuronowej: sieci z jedną warstwą ukrytą i jednym wyjściem, która może zostać użyta do zadania aproksymacji danych. Schemat sieci przedstawia rysunek 5.6. Sieć ta skład się z N neuronów. W każdym z neuronów obliczana jest odpowiedź bloku sumowania s_i:

$$s_i = v_{i0} + \sum_{j=1}^{n} v_{ij} \cdot x_j. \tag{5.30}$$

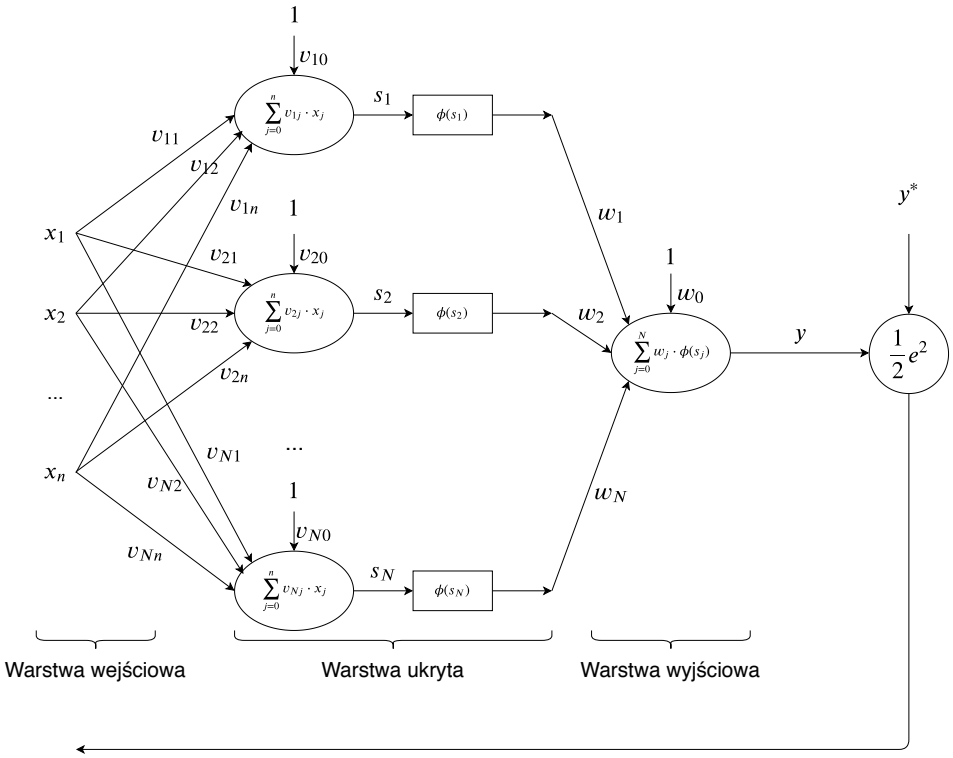
Następnie obliczana jest odpowiedź bloku funkcji aktywacji:

$$\phi(s_i) = \frac{1}{1+e^{-s_i}}. \tag{5.31}$$

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:

$$y = w_0 + \sum_{j=1}^{N} w_j \, \phi(s_j). \tag{5.32}$$

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



Rys. 5.6: Szczegółowy schemat działania sieci neuronowej z jedną warstwą ukrytą. (źródło: *opracowanie własne*)

Wzór na uczenie sieci neuronowej bazuje na wzorze (5.26). Błąd sieci może zostać uproszczony do postaci:

$$\frac{1}{2}e^2 = \frac{1}{2}(y - y^*)^2 \tag{5.33}$$

gdzie y to odpowiedź sieci neuronowej, a y^* to wartość oczekiwana dla danych wejściowych. Znając ogólny wzór na korekcję wag (5.26), można wyprowadzić z niego wzory na korekcję wag w oraz v. W przypadku wag v wyprowadzenie ma

następującą postać:

$$\begin{aligned}\frac{\partial \frac{1}{2}e^2}{\partial v_{ij}} &= \frac{\partial \frac{1}{2}e^2}{\partial y}\cdot\frac{\partial y}{\partial \phi(s_i)}\cdot\frac{\partial \phi(s_i)}{\partial s_i}\cdot\frac{\partial s_i}{\partial v_{ij}} \\ &= \frac{\partial \frac{1}{2}(y-y^*)^2}{\partial y}\cdot\frac{\partial (w_0+w_1\phi(s_1)+\cdots+w_i\phi(s_i)+\cdots+w_N\phi(s_N))}{\partial \phi(s_i)}\cdot \\ &\quad\cdot\frac{\partial \phi(s_i)}{\partial s_i}\cdot\frac{\partial (v_{i0}+v_{i1}x_1+\cdots+v_{ij}x_j+\cdots+v_{in}x_n)}{\partial v_{ij}} \\ &= (y-y^*)\cdot w_i\cdot\phi(s_i)\,(1-\phi(s_i))\cdot x_j.\end{aligned} \tag{5.34}$$

Z kolei wyprowadzenie dla wag w ma postać:

$$\begin{aligned}\frac{\partial \frac{1}{2}e^2}{\partial w_j} &= \frac{\partial \frac{1}{2}e^2}{\partial y}\cdot\frac{\partial y}{\partial w_j} \\ &= \frac{\partial \frac{1}{2}(y-y^*)^2}{\partial y}\cdot\frac{\partial (w_0+w_1\phi(s_1)+\cdots+w_j\phi(s_j)+\cdots+w_N\phi(s_N))}{\partial w_j} \\ &= (y-y^*)\cdot\phi(s_j).\end{aligned} \tag{5.35}$$

W efekcie otrzymujemy dwa wzory na korekcje wag sieci:

$$v_{ij} = v_{ij} - \eta\cdot(y-y^*)\cdot w_i\cdot\phi(s_i)(1-\phi(s_i))\cdot x_j, \tag{5.36}$$

$$w_j = w_j - \eta\cdot(y-y^*)\cdot\phi(s_j). \tag{5.37}$$

5.3.2 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 [25]. 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:

$$\Delta w(t) = -\eta\frac{\partial \frac{1}{2}e^2(t)}{\partial w(t)} + \nu\Delta w(t-1), \tag{5.38}$$

gdzie ν jest tzw. współczynnikiem momentum (ang. *momentum rate*) wybieranym z przedziału $[0,1)$. Zwykle przyjmuje się stosunkowo wysokie wartości ν, np. $\nu = 0.9$ [4, 36, 40].

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 (5.38) 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 [25].

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 [25].

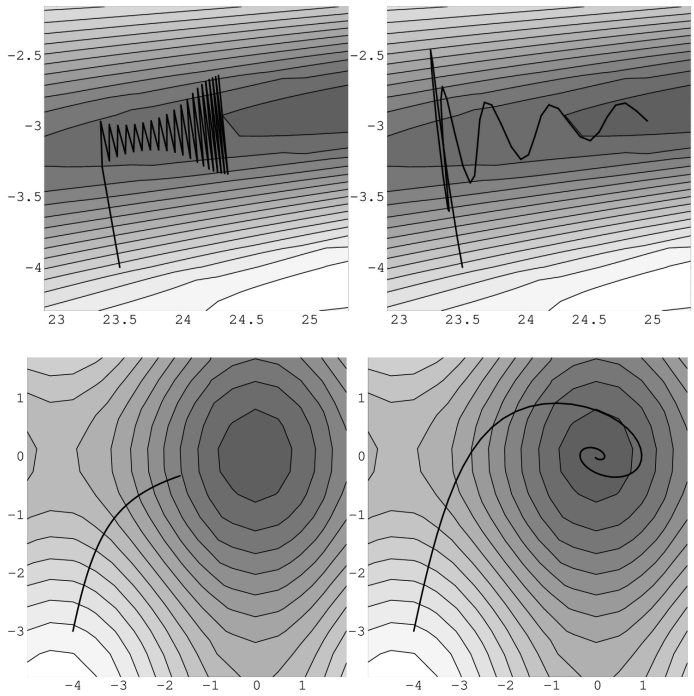
Rekurencyjne rozwinięcie wzoru (5.38) jest następujące:

$$\begin{aligned}
\Delta w(t) &= -\eta \frac{\partial \frac{1}{2}e^2(t)}{\partial w(t)} + \nu \Delta w(t-1) \\
&= -\eta \frac{\partial \frac{1}{2}e^2(t)}{\partial w(t)} + \nu \left(-\eta \frac{\partial \frac{1}{2}e^2(t-1)}{\partial w(t)} + \nu \Delta w(t-2) \right) \\
&= -\eta \frac{\partial (\frac{1}{2}e^2(t))}{\partial w(t)} + \nu \left(-\eta \frac{\partial \frac{1}{2}e^2(t-1)}{\partial w(t-1)} + \nu \left(-\eta \frac{\partial \frac{1}{2}e^2(t-2)}{\partial w(t-2)} + \cdots \right) \right) \\
&= -\eta \sum_{k=0}^{\infty} \nu^k \frac{\partial \frac{1}{2}e^2(t-k)}{\partial w(t-k)}.
\end{aligned} \tag{5.39}$$

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 [40].

Przykładowe ilustracje porównujące działanie zwykłej metody gradientowej z metodą momentum pokazano na Rys. 5.7.

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



Rys. 5.7: 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: (25))

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

$$\begin{aligned}\Delta w(t) &= -\eta \sum_{k=0}^{\infty} \nu^k \frac{\partial \frac{1}{2}e^2(t-k)}{\partial w(t-k)} = -\eta G \sum_{k=0}^{\infty} \nu^k \\ &= -\frac{\eta}{1-\nu} G. \end{aligned} \tag{5.40}$$

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 [36].

5.3.3 Resilient Backpropagation — RPROP

Metodę RPROP (ang. *Resilient backPROPagation*) zaproponowali w 1993 r. Riedmiller i Braun [41]. 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:

$$w(t+1)=w(t)-\eta_w(t)\,\mathrm{sgn}\,\frac{\partial \frac{1}{2}E^2(t)}{\partial w(t)}. \tag{5.41}$$

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.:

$$\frac{1}{2}E^2(t)=\sum_{i=1}^{m}\frac{1}{2}e_i^2(t). \tag{5.42}$$

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

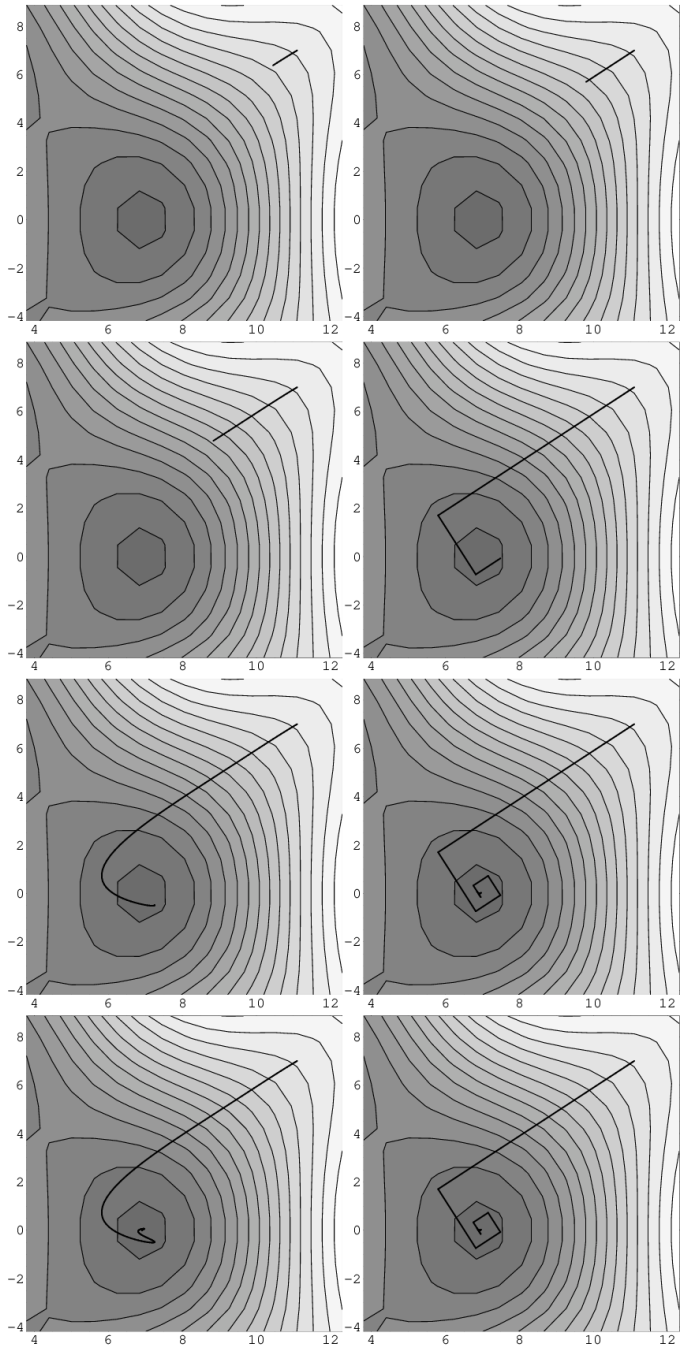
Bardzo istotnym elementem metody jest wzór na modyfikację każdego współczynnika uczenia, który ma postać:

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

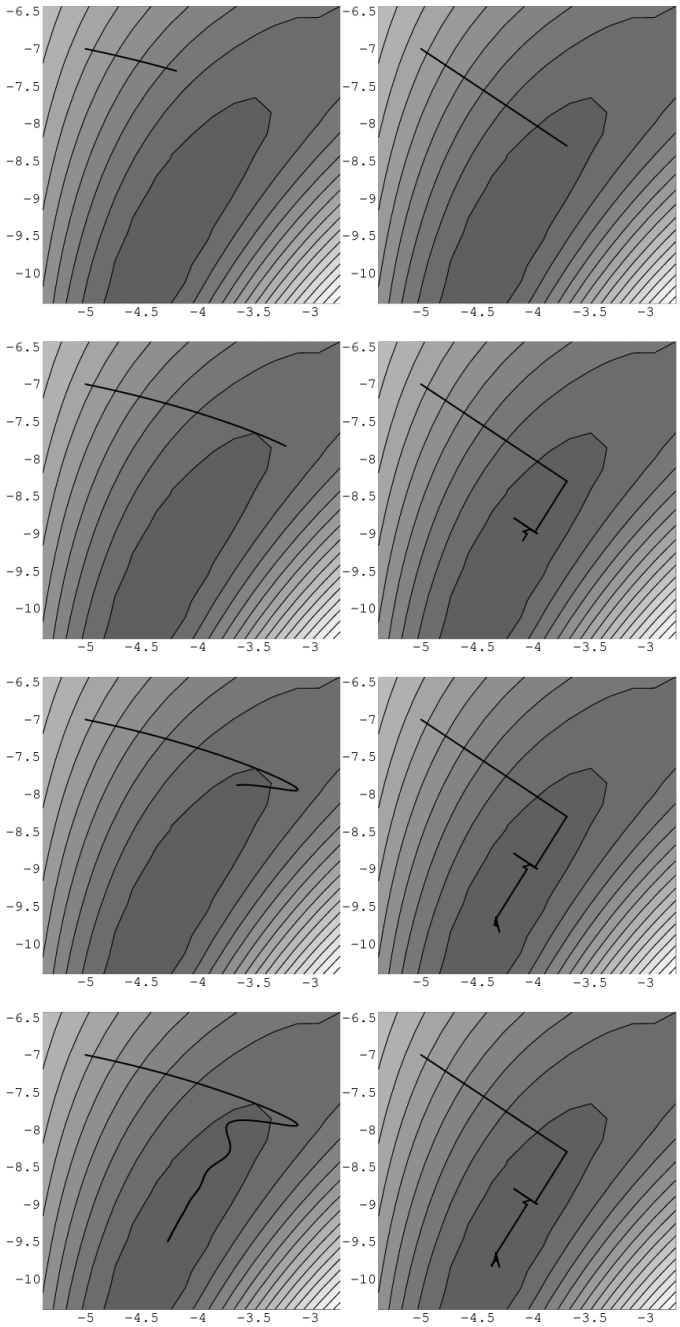
Zwyczajowo przyjmuje się następujące wartości dla stałych występujących w tym wzorze: $a = 1.2$ (tempo zwiększania współczynnika uczenia), $b = 0.5$ (tempo zmniejszania współczynnika uczenia), $\eta_{\min} = 10^{-6}$, $\eta_{\max} = 10^2$ (skrajne wartości współczynnika uczenia). W chwili startu można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

Rysunki 5.8 i 5.9 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. 5.8: 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: (25))



Rys. 5.9: 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: (25))

5.4 Ćwiczenia laboratoryjne (MATLAB)

E **Ćwiczenie 5.1** **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.

E **Ćwiczenie 5.2** **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.

E **Ćwiczenie 5.3** **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 5.4 **Przebadaj działanie sieci dla różnej liczby neuronów w warstwie ukrytej (wykorzystaj program z Ćwiczenia 5.3).** 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 5.5 **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 5.3, 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).

6. 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.

6.1 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*

6.1.1 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:

$$P(A|B) = \frac{P(A \cap B)}{P(B)}, \tag{6.1}$$

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 —

$$P(A,B|C) = \frac{P(A,B,C)}{P(C)} = \frac{P(A,B,C) \cdot P(B,C)}{P(C) \cdot P(B,C)} = P(A|B,C)P(B|C). \tag{6.2}$$

– przenoszenie zdarzenia B przed kreskę warunkowania —

$$P(A|B,C) = \frac{P(A,B,C)}{P(B,C)} = \frac{P(A,B,C) \cdot P(C)}{P(B,C) \cdot P(C)} = \frac{P(A,B|C)}{P(B|C)}. \tag{6.3}$$

6.1.2 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 6.1.1 — 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:

$$P(A \cap B) = P(A) \cdot P(B). \tag{6.4}$$

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:

$$P(A|B) = \frac{P(A \cap B)}{P(B)} = \frac{P(A) \cdot P(B)}{P(B)} = P(A). \tag{6.5}$$

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 (6.4) musiałby zachodzić dla wszystkich możliwych podstawień par wartości do tych zmiennych, tj.:

$$\forall_{h \in \{\mathrm{m},\acute{\mathrm{s}},\mathrm{d}\}} \forall_{c \in \{\mathrm{z},\mathrm{n},\mathrm{b},\mathrm{s}\}} \quad P(H = h \cap C = c) = P(H = h) \cdot P(C = c). \tag{6.6}$$

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.

6.1.3 Prawdopodobieństwo całkowite

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

Twierdzenie 6.1.1 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:

$$\begin{aligned} P(A) &= P(A|B_1)P(B_1) + P(A|B_2)P(B_2) + \cdots + P(A|B_n)P(B_n) \\ &= \sum_{i=1}^{n} P(A|B_i)P(B_i). \end{aligned} \tag{6.7}$$

Dowód. Idąc od prawej strony wzory (6.7) 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 (6.7). 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 $P(B_i|A)$:

$$\begin{aligned} P(B_i|A) &= \frac{P(B_i \cap A)}{P(A)} \\ &= \frac{P(B_i \cap A)}{P(A|B_1)P(B_1) + \cdots + P(A|B_n)P(B_n)} \\ &= \frac{P(B_i \cap A)\frac{P(B_i)}{P(B_i)}}{P(A|B_1)P(B_1) + \cdots + P(A|B_n)P(B_n)} \\ &= \frac{P(A|B_i)P(B_i)}{P(A|B_1)P(B_1) + \cdots + P(A|B_n)P(B_n)}. \end{aligned} \tag{6.8}$$

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

6.2 Naiwny klasyfikator Bayesa

6.2.1 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.:

$$\forall y \, \forall i \neq j \qquad X_i|Y = y \perp X_j|Y = y. \tag{6.9}$$

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

[2]Przekleństwo wymiarowości (ang. *curse of dimensionality*) — zjawisko występując w niektó-

6.2.2 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).

Przypuśćmy, że do dyspozycji jest pewien zbiór danych uczących z dyskretnymi zmiennymi wejściowymi X_i, $i = 1,\ldots,n$, oraz z wyróżnioną 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 w Tab. 6.1.

Tabela 6.1: 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

rych 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.

[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:

$$y^* = \underset{y\in\{1,\ldots,K\}}{\arg\max}\, P(Y = y|X = x). \tag{6.10}$$

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

$$\begin{aligned} P(Y = y|X = x) &= \frac{P(X = x|Y = y)P(Y = y)}{P(X)} \\ &= \frac{P(X = x|Y = y)P(Y = y)}{P(X = x|Y = 1)P(Y = 1) + \cdots + P(X = x|Y = K)P(Y = K)}. \end{aligned} \tag{6.11}$$

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:

$$y^* = \underset{y\in\{1,\ldots,K\}}{\arg\max}\, P(Y = y|X = x) = \underset{y\in\{1,\ldots,K\}}{\arg\max}\, P(X = x|Y = y)P(Y = y). \tag{6.12}$$

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

$$\begin{aligned} P(X = x|Y = y) &= P(X_1 = x_1 \cap X_2 = x_2 \cap \cdots \cap X_n = x_n|Y = y) \\ &= P(X_1 = x_1|Y = y)P(X_2 = x_2|Y = y)\cdots P(X_n = x_n|Y = y) \\ &= \prod_{j=1}^{n} P(X_j = x_j|Y = y). \end{aligned} \tag{6.13}$$

A zatem, wychodząc od (6.12), interesujący nas końcowy **wzór dla wariantu dyskretnego**, pozwalający przyporządkować obiektowi $x = (x_1, x_2, \ldots, x_n)$ najbardziej prawdopodobną klasę, przyjmuje postać:

$$y^* = \underset{y\in\{1,\ldots,K\}}{\arg\max} \prod_{j=1}^{n} P(X_j = x_j|Y = y)P(Y = y). \tag{6.14}$$

Prosty przykład obliczeń dla NBC ze zmiennymi dyskretnymi

W ramach przećwiczenia działania NBC i wzoru (6.14) 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. 6.2 — 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 6.2: 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 (6.14). Utożsamiając prawdopodobieństwa z częstościami występującymi

w Tab. 6.2, byłyby to następujący zestaw:

$$P(Y=-)=7/16$$
$$P(X_1=M|Y=-)=1/7 \qquad P(X_1=K|Y=-)=6/7$$
$$P(X_2=-|Y=-)=3/7 \qquad P(X_2=+|Y=-)=4/7$$
$$P(X_3=-|Y=-)=6/7 \qquad P(X_3=+|Y=-)=1/7$$

$$P(Y=+)=9/16$$
$$P(X_1=M|Y=+)=6/9 \qquad P(X_1=K|Y=+)=3/9$$
$$P(X_2=-|Y=+)=6/9 \qquad P(X_2=+|Y=+)=3/9$$
$$P(X_3=-|Y=+)=4/9 \qquad P(X_3=+|Y=+)=5/9$$

Sklasyfikujemy teraz dwa przykładowe nowo przychodzące obiekty: $(M,-,+)$ oraz $(K,-,-)$. Wzór (6.14) 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

$$\begin{aligned}P(X_1=M|Y=-)\cdot P(X_2=-|Y=-)\cdot P(X_3=+|Y=-)\cdot P(Y=-)\\ =\frac{1}{7}\cdot\frac{3}{7}\cdot\frac{1}{7}\cdot\frac{7}{16}=\frac{3}{784}\approx 0.0038265,\end{aligned}$$

zaś ustalając klasę $y=+$ analogiczny iloczyn to

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

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 (6.11)), 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:

$$P\big(Y=-|X=(M,-,+)\big)=\frac{\frac{3}{784}}{\frac{3}{784}+\frac{1620}{11664}}\approx 0.0268123,$$
$$P\big(Y=+|X=(M,-,+)\big)=\frac{\frac{1620}{11664}}{\frac{3}{784}+\frac{1620}{11664}}\approx 0.9731877.$$

Postępując analogicznie dla drugiego obiektu $(K,-,-)$, można przekonać się, że interesujące nas iloczyny wynoszą odpowiednio $108/343$ i $8/81$, które po znormalizowaniu do prawdopodobieństw przełożyłyby sie na:

$$P\big(Y=-|X=(K,-,-)\big)=\frac{\frac{108}{343}}{\frac{108}{343}+\frac{8}{81}}\approx 0.7612252,$$
$$P\big(Y=+|X=(K,-,-)\big)=\frac{\frac{8}{81}}{\frac{108}{343}+\frac{8}{81}}\approx 0.2387748.$$

Patrząc ponownie na główny wzór (6.14), 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 (6.14). 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

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

na 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 (6.14), to łatwo stwierdzić, że jest ona klasy $O(n)$ — czyli liniowa ze względu 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 danych 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 $P(X_i=v|Y=y)$, zaś wystarcza dokładnie jeden przebieg po danych. Dla przykładu, biegnąc po pierwszym wierszu Tab. 6.2, 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 liczniki należy zamienić na prawdopodobieństwa, wykonując dzielenia dla poszczególnych klas zgodnie ze wzorem na prawdopodobieństwo warunkowe (6.1). 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: $O(n)$.

[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 (6.14). Co, jeśli którykolwiek z czynników w tym wzorze byłby równy 0? Oczywiście, spowodowałoby to wyzerowanie całego wyniku niezależnie od tego, czy pozostałe czynniki były przeciętnie niskie czy też 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). Jednym 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

$$P(A) \approx \frac{k+1}{m+q}. \tag{6.15}$$

W szczególności dla zdarzeń binarnych powyższy wzór wynosi $\frac{k+1}{m+2}$.

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 $O(Kn\bar{q})$. 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.

6.2.3 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 (6.14) 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:

$$P(a \leqslant X \leqslant b) = \int_a^b p(x)\,dx. \tag{6.16}$$

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:

$$F(a) = P(X \leqslant a) = \int_{-\infty}^{a} p(x)\,dx. \tag{6.17}$$

Można zapisać następujące związki pomiędzy gęstością a dystrybutantą:

– różniczka dystrybuanty = gęstość · przyrost:

$$dF(x) = p(x)\,dx, \tag{6.18}$$

– całka nieoznaczona z funkcji gęstości = dystrybuanta + stała:

$$\int p(x)\,dx = F(x) + C \tag{6.19}$$

– prawdopodobieństwo jako przyrost dystrybuanty:

$$P(a \leqslant X \leqslant b) = \int_a^b p(x)\,dx = F(b) - F(a). \tag{6.20}$$

Interesujący nas **wzór dla wariantu ciągłego** naiwnego klasyfikatora Bayesa to „kuzyn" wzoru (6.14), w którym w miejsce prawdopodobieństw wpisujemy wartości warunkowych funkcji *gęstości* w punkcie (z wyjątkiem prawdopodobieństw a priori klas — te pozostają bez zmian):

$$y^* = \underset{y\in\{1,\ldots,K\}}{\arg\max} \prod_{j=1}^{n} p_j(x_j|Y=y)P(Y=y). \tag{6.21}$$

Należy być świadomym następujących ograniczeń związanych ze wzorem (6.21):

1. zwracana przezeń wartość nie powinna być interpretowana jako prawdopodobieństwo, nawet po normalizacji, ze względu na mieszane czynniki p i P o różnym sensie probabilistycznym (wzór jedynie „przypomina" iloczyn prawdopodobieństw),
2. nadal w mocy jest założenie naiwne — wyprowadzeniu wzoru (6.21) (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 (6.21) 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

$$p(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}, \tag{6.22}$$

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

$$p_j(x|Y = y) = \frac{1}{\sigma_{jy}\sqrt{2\pi}} e^{-\frac{(x-\mu_{jy})^2}{2\sigma_{jy}^2}}, \tag{6.23}$$

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

$$\mu_{jy} = \frac{1}{m} \sum_{\substack{i=1 \\ y_i=y}}^{m} x_{ij}, \tag{6.24}$$

$$\sigma_{jy} = \sqrt{\frac{1}{m-1} \sum_{\substack{i=1 \\ y_i=y}}^{m} (x_{ij} - \mu_{jy})^2}. \tag{6.25}$$

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 (6.24) 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 $\frac{1}{m}$.

6.2.4 Przykłady działania NBC

Tab. 6.3 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 6.3: 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”	$100 \times 2, 2$	86.67%	84.00%	85.33%	92.00%	93.33%	100.00%	84.00%	92.00%
„wine”	$178 \times 13, 3$	92.48%	100.00%	97.74%	95.56%	96.99%	100.00%	97.74%	95.56%
„spambase”	$4601 \times 57, 2$	68.20%	67.51%	78.26%	77.76%	82.23%	82.19%	82.06%	83.93%
„iris”	$150 \times 4, 3$	94.64%	97.37%	93.75%	92.11%	90.18%	86.84%	94.64%	97.37%
„sonar”	$208 \times 60, 2$	80.13%	76.92%	88.46%	76.92%	86.54%	84.62%	73.08%	71.15%

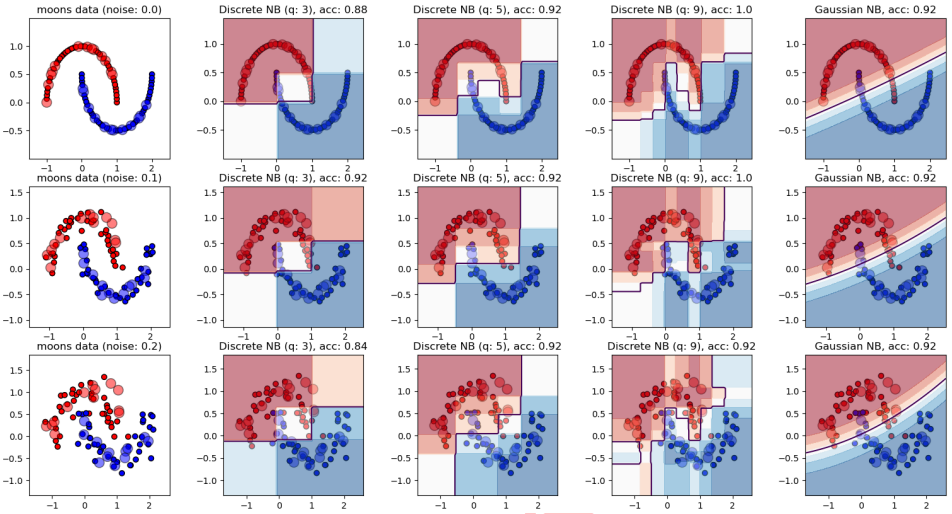
W drugiej kolumnie tabeli podano pełne rozmiary zbiorów — liczba przykładów $\times$ liczba zmiennych (cech) — oraz po przecinku liczbę klas decyzyjnych. Każdy zbiór danych został wstępnie podzielony na część uczącą i testową w proporcji 75% : 25%. Do przygotowania klasyfikatorów (czyli wyznaczenia i zapamiętania potrzebnych rozkładów prawdopodobieństwa) użyto tylo 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. 6.1 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).

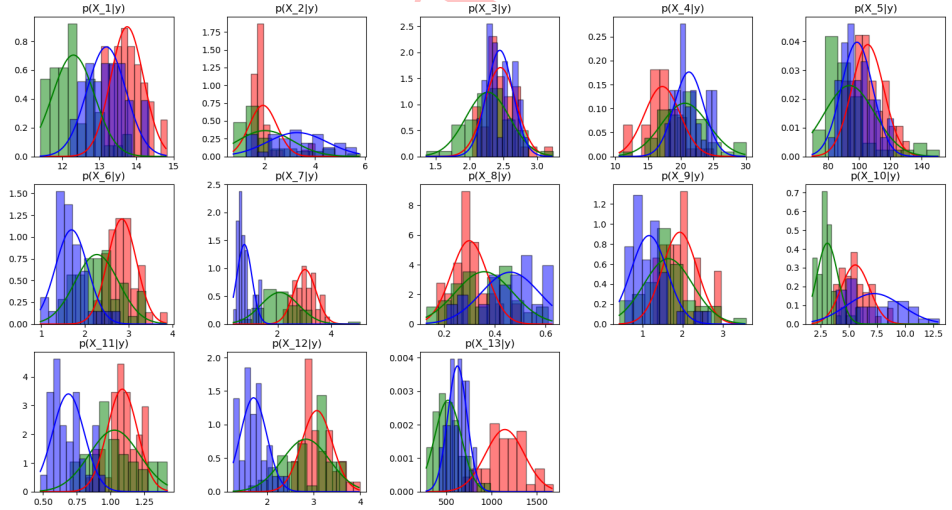
Dla lepszego zrozumienia różnic w podejściu do estymowania rozkładów pomiędzy dyskretnymi a gaussowskimi NBC przygotowano rysunki 6.2 i 6.3. Dotyczą

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

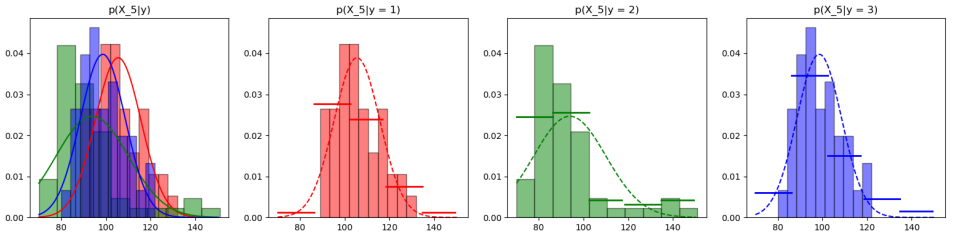
[8] Wyniki przedstawione w tabeli dla zbioru „moons” dotyczą wariantu tego zbioru wygenerowanego z parametrami `noise=0.1` oraz `random_state=0`. Wywołanie: `sklearn.datasets.make_moons(noise=0.1, random_state=0)`.



Rys. 6.1: 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. 6.2: Histogramy i przybliżenia normalne dla warunkowych rozkładów prawdopodobieństwa zmiennych w danych „wine". (źródło: *opracowanie własne*)



Rys. 6.3: Rozkład zmiennej nr 5 w danych „wine” — porównanie: przybliżenia normalne vs. przybliżenia kawałkami stałe (przy dyskretyzacji na 5 równoszerokich przedziałów). (źródło: *opracowanie własne*)

one zbioru „wine” (rozpoznawanie gatunków wina) i przedstawiają: histogramy wszystkich rozkładów warunkowych, ich przybliżenia normalne, a także przybliżenia kawałkami stałe dla wybranej zmiennej będące równoważne dyskretyzacji.

6.2.5 Bezpieczeństwo numeryczne obliczeń NBC

Wzory wyznaczające odpowiedź NBC — (6.14) i (6.21) — 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 (6.14), (6.21) można zlogarytmować, zamieniając tym samym iloczyn na sumę, co nie wpłynie na podejmowaną decyzję ze względu na monotoniczność. Logarytmy prawdopodobieństw są licz-

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

bami ujemnymi[11], a więc czytelnik może zastanawiać się, kiedy suma odpowiednio wielu ujemnych składników również doprowadzi do niedomiaru i osiągnie wartość `-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 (6.14) dla NBC ze zmiennymi dyskretnymi przyjmuje następującą postać po zlogarytmowaniu:

$$y^* = \arg\max_{y\in\{1,\ldots,K\}} \sum_{j=1}^{n} \log P(X_j = x_j | Y = y) + \log P(Y = y). \tag{6.26}$$

Z kolei odpowiednik wzoru (6.21) dla NBC ze zmiennymi ciągłymi modelowanymi poprzez rozkłady normalne o funkcji gęstości (6.23) przyjmuje następującą postać po zlogarytmowaniu:

$$y^* = \arg\max_{y\in\{1,\ldots,K\}} \sum_{j=1}^{n} \left(\log 1 - \log \sigma_{jy} - \log \sqrt{2\pi} + \log e^{-\frac{(x-\mu_{jy})^2}{2\sigma_{jy}^2}} \right) + \log P(Y = y) \tag{6.27}$$

$$= \arg\max_{y\in\{1,\ldots,K\}} \sum_{j=1}^{n} \left(-\log \sigma_{jy} - \frac{(x-\mu_{jy})^2}{2\sigma_{jy}^2} \right) + \log P(Y = y). \tag{6.28}$$

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.

[11] niekoniecznie logarytmy funkcji gęstości

6.3 Ćwiczenia laboratoryjne (Python)

Ćwiczenie 6.1 **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 (6.14) 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.

Ćwiczenie 6.2 **Napisz program realizujący NBC w wersji ciągłej dla zbioru „wine" z repozytorium UCI** Jako rozszerzenie ćwiczenia 6.1 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.

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

Porównaj także zgodność działania otrzymanego klasyfikatora (Twojej implementacji) z gotową implementacją `GaussianNB` dostępną w pakiecie `sklearn.naive_bayes`.

E **Ćwiczenie 6.3 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 6.4 Zmodyfikuj opracowane implementacje NBC zapewniając bezpieczeństwo numeryczne obliczeń** Wykorzystując zabieg logarytmowania, zmodyfikuj implementacje opracowane na rzecz ćwiczeń 6.1 i 6.2, tak aby obliczanie odpowiedzi klasyfikatora było zgodne odpowiednio z wzorami (6.26) i (6.28). 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).

7. Podstawy Statystycznej Teorii Uczenia

Ostatnie dwie 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" [54] (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.

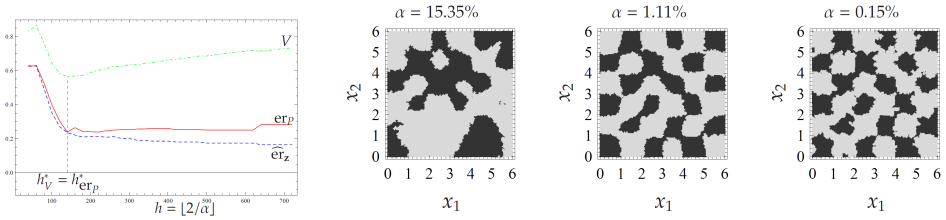
Pomimo tej swoistej trudności, istnieją pewne rezultaty matematyczne, które

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 [26]. Wspomniane rezultaty zostały sformułowane w ramach tzw. ***Statystycznej Teorii Uczenia*** (ang. *Statistical Learning Theory*) zapoczątkowanej przez Vladimira Vapnika [51, 52, 53] z wykorzystaniem techniki matematycznej ***PAC*** (ang. *Probably Approximately Correct*) pochodzącej od Valianta [50].

SLT i PAC zwracają szczególną uwagę na relację pomiędzy rozmiarem próby uczącej a złożonością przyjętego modelu, i na tej podstawie dostaraczają 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. 7.1 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 szumów w danych uczących.

[1] zwykle w postaci probabilistycznych ograniczeń (nierówności)



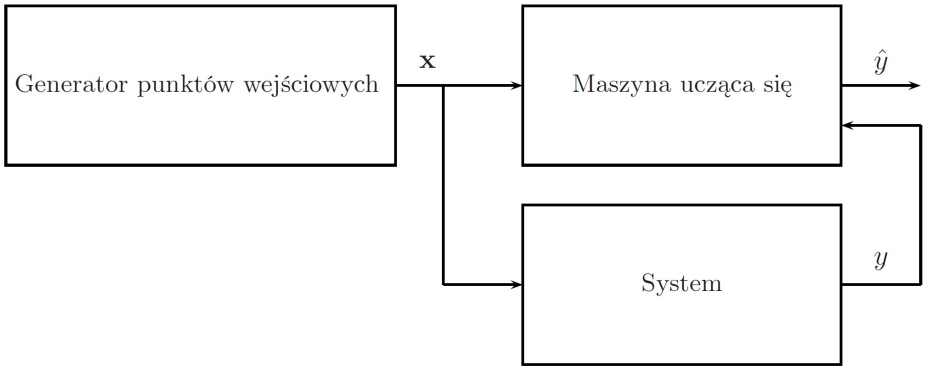
Rys. 7.1: Klasyfikacja wzorca „szachownica" z wykorzystaniem algorytmu najbliższych sąsiadów (źródło: (24)). Wykres po lewej stronie pokazuje przebieg procedury wyboru złożoności modelu (złożoność określona przez parametr α — procent najbliższych sąsiadów), gdzie obserwowane są: błąd na próbie, błąd prawdziwy, i ograniczenie na błąd oparte na wymiarze VC. 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.

7.1 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. 7.2. 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. 7.2: Maszyna ucząca się na podstawie obserwacji systemu (źródło: *opracowanie własne* na podstawie (1, 7, 40)).

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 [7].

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

[7, 12, 25].

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

$$f(x;\mathbf{w},\mathbf{v}) = w_0 + \sum_{k=1}^{N} \left(v_k \sin(kx) + w_k \cos(kx)\right), \tag{7.1}$$

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

$$f(x;\mathbf{w}) = \sum_{k=0}^{N} w_k x^k. \tag{7.2}$$

Natomiast zbiór zawierająca funkcje typu

$$f(\mathbf{x};\mathbf{w},\mathbf{V}) = w_0 + \sum_{k=1}^{N} w_k \phi\left(v_{k0} + \sum_{j=1}^{n} v_{kj} x_j\right), \tag{7.3}$$

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 [25], o których była mowa w punkcie 5.2.

Rozróznienie 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 [7, 12, 25].

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. [1, 51, 52]). Duża część spośród następujących treści jest powtórzona za pracą [26].

7.2 Notacja i pojęcia podstawowe

Niech

$$\mathbf{z} = \{\mathbf{z}_1, \mathbf{z}_2, \ldots, \mathbf{z}_m\} = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_m, y_m)\}. \tag{7.4}$$

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) [26]. Dla uproszczenia pominiemy w dalszych rozważaniach zadania estymacji gęstości i klasteryzacji, które są zadaniami uczenia nienadzorowanego (ang. *unsupervised learning*)[3].

Niech

$$F = \{f\}, \tag{7.5}$$

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 [26]. Chodzi tu o błąd policzony w sposób dokładny jako wartość oczekiwana względem rozkładu P. Dla zadania klasyfikacji błąd prawdziwy definiujemy jako:

$$\mathrm{er}_P(f) = \int\limits_{\mathbf{x}\in\mathbf{X}} \sum_{y\in Y} [f(\mathbf{x}) \neq y] \underbrace{p(\mathbf{x})P(y|\mathbf{x})\mathbf{dx}}_{dP(\mathbf{x},y)}, \tag{7.6}$$

[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 [51, 52].

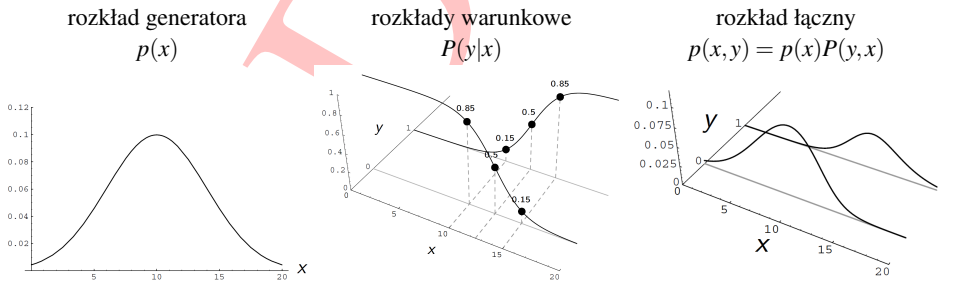
gdzie notacja $[\cdot]$ jest funkcją wskaźnikową, przyjmującą 1 gdy zdanie będące argumentem jest prawdziwe i 0 w przeciwnym razie. Jak można zauważyć, liczbowy sens er_P to *prawdopodobieństwo błędnego sklasyfikowania* losowej pary $(\mathbf{x}, y)$ zaczerpniętej z P. Dla zadania estymacji funkcji regresji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

$$\mathrm{er}_P(f) = \int_{\mathbf{x}\in\mathbf{X}} \int_{y\in Y} (f(\mathbf{x}) - y)^2 \underbrace{p(\mathbf{x})p(y|\mathbf{x})dy\mathbf{dx}}_{dP(\mathbf{x},y)}. \tag{7.7}$$

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. 7.3 obrazuje rozkłady składowe (brzegowe) oraz łączny prawdopodobieństwa definiujące ten problem. Pierwszy z wykresów przedstawia rozkład generatora określony przez gęstość $p(x)$,



Rys. 7.3: 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

[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).

warunkowe rozkłady prawdopodobieństwa $P(y|x)$, zgodnie z którymi objawiają 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. 7.3 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:

$$p(x) = \frac{1}{\sqrt{2\pi}4.0} e^{-\frac{(x-10.0)^2}{2\cdot 4.0^2}},$$
$$P(y=0|x) = \frac{1}{1+e^{0.75(x-12.5)}}, \quad P(y=1|x) = 1 - \frac{1}{1+e^{0.75(x-12.5)}}. \tag{7.8}$$

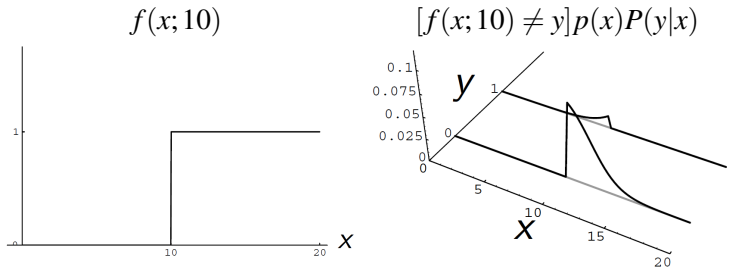
Dla powyższego przykładu rozważmy teraz maszynę uczącą się, która ma do dyspozycji następujący zbiór funkcji $F = \{f(x;\omega)\}_{\omega\in\mathbb{R}}$, gdzie

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

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.

[7] continuum rozkładów

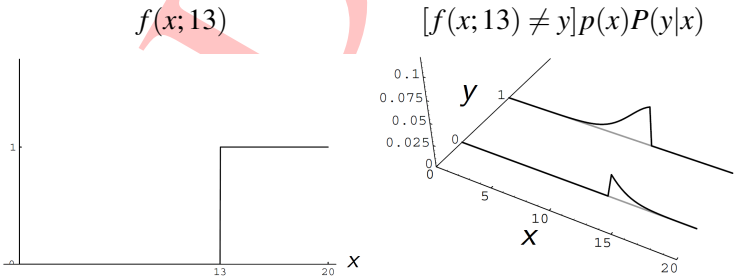


Rys. 7.4: Wykres funkcji $f(x;10)$ oraz jej funkcji straty ważonej łącznym rozkładem prawdopodobieństwa (źródło: *opracowanie własne*).

Rys. 7.4 przedstawia wykres funkcji $f(x;10)$ oraz jej funkcji straty ważonej rozkładem łącznym. To właśnie ta ważona funkcja straty będzie całkowana 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

$$\int_{-\infty}^{\infty} \sum_{y\in\{0,1\}} [f(x;10) \neq y]p(x)P(y|x)\,dx \approx 0.239671. \tag{7.9}$$

Postępując analogicznie dla funkcji $f(x;13)$ możemy wyznaczyć jej ważoną funkcję straty (patrz Rys. 7.5) i tym samym obliczyć poprzez całkowanie jej



Rys. 7.5: Wykres funkcji $f(x;13)$ oraz jej funkcji straty ważonej łącznym rozkładem prawdopodobieństwa (źródło: *opracowanie własne*).

błąd prawdziwy:

$$\int_{-\infty}^{\infty} \sum_{y\in\{0,1\}} [f(x;10) \neq y]p(x)P(y|x)\,dx \approx 0.143819. \tag{7.10}$$

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 (7.8). 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 (7.4). 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

$$\widehat{\mathrm{er}}_{\mathbf{z}}(f) = \frac{1}{m}\sum_{i=1}^{m}[f(\mathbf{x}_i) \neq y_i], \tag{7.11}$$

natomiast dla zadania estymacji funkcji regresji jako

$$\widehat{\mathrm{er}}_{\mathbf{z}}(f) = \frac{1}{m}\sum_{i=1}^{m}(f(\mathbf{x}_i) - y_i)^2. \tag{7.12}$$

Liczbowy sens (7.11) i (7.12) to odpowiednio *częstość błędnej klasyfikacji* oraz *średni kwadrat odchyłki* [26].

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:

$$\widehat{f} = \arg\inf_{f\in F} \widehat{\mathrm{er}}_{\mathbf{z}}(f). \tag{7.13}$$

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

$$L\colon \bigcup_{m=1}^{\infty}(\mathbf{X}\times\mathbf{Y})^m \to F, \tag{7.14}$$

[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.

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 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 *back-propagation*, algorytmem *RPROP*, 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. [26]

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:

$$f^* = \arg \inf_{f \in F} \mathrm{er}_P(f). \tag{7.15}$$

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:

$$l_F = \{l_f \colon\ f \in F\}, \tag{7.16}$$

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 re-

gresji mamy funkcje $l_f(\mathbf{z}) = (f(\mathbf{x}) - y)^2$ realizujące odwzorowanie $l_f\colon\ \mathbf{X} \times Y \to \mathbb{R}$. Pomiędzy zbiorami 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.

7.3 Zbieżność jednostajna i pojęcia złożoności maszyn uczących się

7.3.1 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 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:

$$P_m(|p - \nu_m| > \varepsilon) \leqslant 2e^{-2\varepsilon^2 m}, \tag{7.17}$$

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:

$$P_m(p - \nu_m > \varepsilon) \leqslant e^{-2\varepsilon^2 m}, \tag{7.18}$$

$$P_m(\nu_m - p > \varepsilon) \leqslant e^{-2\varepsilon^2 m}. \tag{7.19}$$

[10] Wwaga: nie należy w tym kontekście odczytywać oznaczenia p jako funkcji gęstości.

Rozważmy najprostszy przypadek *skończonego* zbioru funkcji $F = \{f_1, \ldots, f_N\}$ użytego do uczenia. Znany jest następujący elementarny rezultat [52, 53] o jednostajnej zbieżności:

$$P_m\left(\sup_{f\in F}(\mathrm{er}_P(f) - \widehat{\mathrm{er}}_{\mathbf{z}}(f)) > \varepsilon\right) \leqslant \sum_{k=1}^{N} P_m\left(\mathrm{er}_P(f_k) - \widehat{\mathrm{er}}_{\mathbf{z}}(f_k) > \varepsilon\right) \leqslant N \cdot e^{-2\varepsilon^2 m}, \tag{7.20}$$

gdzie ostatnie przejście wynika z faktu, że dla każdej ustalonej funkcji f_k zachodzi nierówność Chernoffa (7.18)[11]. Przypisując do prawej strony nierówności (7.20) pewne małe prawdopodobieństwo δ i rozwiązując ze względu na ε, powyższy rezultat można równoważnie wyrazić w formie ***ograniczenia na błąd prawdziwy***[12]:

$$\mathrm{er}_P(f_k) \leqslant \widehat{\mathrm{er}}_{\mathbf{z}}(f_k) + \sqrt{\frac{\ln N - \ln \delta}{2m}}, \tag{7.21}$$

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 $1-2\delta$:

$$\mathrm{er}_P(\widehat{f}\,) - \mathrm{er}_P(f^*) \leqslant \sqrt{\frac{\ln N - \ln \delta}{2m}} + \sqrt{\frac{-\ln \delta}{2m}}, \tag{7.22}$$

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 [26].

7.3.2 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 (7.22)[14]. I tak dla uproszczonego przypadku skończonego zbioru

[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.

funkcji złożoność próbkowa jest ograniczona następująco:

$$m_L(\varepsilon,\delta) \leqslant \frac{1}{2\varepsilon^2}\left(\sqrt{\ln N - \ln(\delta)} + \sqrt{-\ln(\delta)}\right)^2. \tag{7.23}$$

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*).

7.3.3 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.:

$$(l_F)_{|\mathbf{z}_1,\ldots,\mathbf{z}_m} = \left\{\big(l_f(\mathbf{z}_1),\ldots,l_f(\mathbf{z}_m)\big):\ f\in F\right\}. \tag{7.24}$$

Oczywiście $\#(l_F)_{|\mathbf{z}_1,\ldots,\mathbf{z}_m} \leqslant 2^m$.

Ważnym pojęciem w tym kontekście jest ***roztrzaskiwanie*** (ang. *shattering*).

Definicja 7.3.1 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.

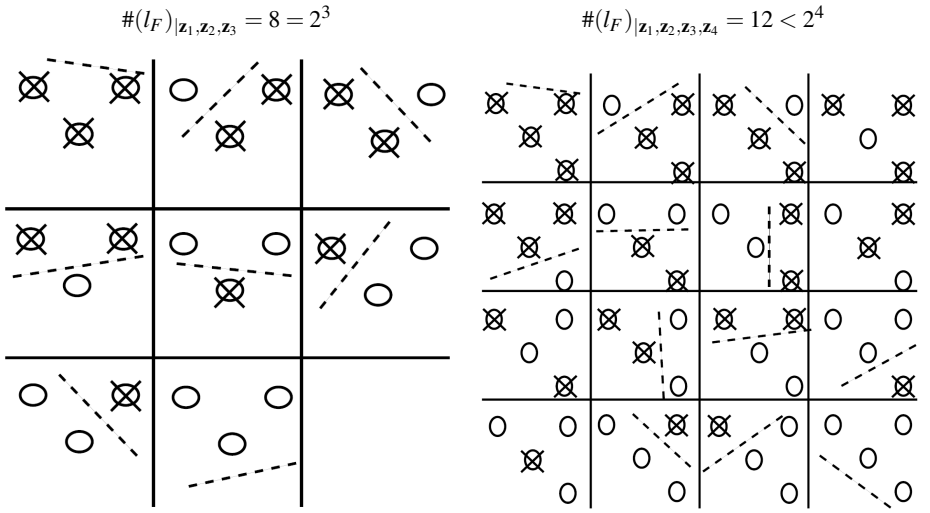
Inaczej mówiąc, oznacza to, że można zrealizować wszystkie dychotomie próby za pomocą funkcji z tego zbioru.

Idąc dalej, za pomocą pojęcia roztrzaskiwania można w poniższy sposób zdefiniować ***wymiar Vapnika-Chervonenkisa*** [53].

Definicja 7.3.2 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. 7.6 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. 7.6: 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. w sieciach perceptronowych, wymiar VC wynosi $n+1$ [52]. Dla kostek w $\mathbb{R}^n$ wymiar VC wynosi $2n$ [7]. Dla kul w $\mathbb{R}^n$, które mogą być bazami dla radialnych sieci neuronowych, wymiar VC wynosi $n+1$ [7]. Dla wielomianów zdefiniowanych nad $\mathbb{R}^n$ stopnia co najwyżej s, wymiar VC wynosi $\binom{s+n}{n}$, patrz np. [1]. Jeżeli chodzi o liniowe kombinacje baz, to wymiar VC można zwykle ograniczyć z góry przez iloczyn liczby baz i wymiaru VC pojedynczej bazy [1, 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 [1, 26].

Alternatywnie, wymiar VC można definiować w oparciu o ***funkcję wzrostu***

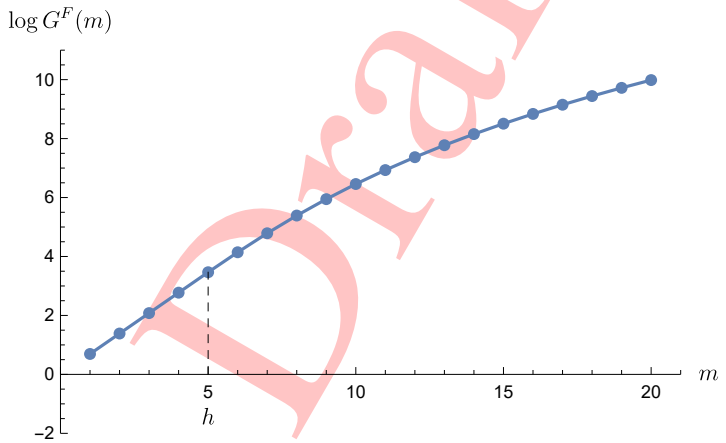
(ang. *growth function*), która podaje największą liczbę rozróżnialnych funkcji dla danego rozmiaru próby:

$$G^F(m) = \sup_{\mathbf{z}\in(\mathbf{X}\times Y)^m} \#(l_F)_{|\mathbf{z}}. \tag{7.25}$$

Wymiar VC, to największy argument funkcji wzrostu, powyżej którego przestaje ona narastać wykładniczo. Jednym z istotnych rezultatów w tym kontekście jest lemat Sauera [6, 46, 49], który mówi, że jeżeli VC-dim$(l_F) = h$, to:

$$G^F(m) \leqslant \sum_{i=0}^{h} \binom{m}{h} < \left(\frac{me}{h}\right)^h. \tag{7.26}$$

Na Rys. 7.7 przedstawiono przykładowy wykres zlogarytmowanej funkcji wzrostu przy ustaleniu wymiaru VC na wartość $h = 5$.



Rys. 7.7: 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 [53], patrz także [1].

Twierdzenie 7.3.1 Niech F będzie nieskończonym zbiorem funkcji

zero-jedynkowych o funkcji wzrostu $G^F(m)$ i VC-dim$(F) = h$. Wówczas:

$$P_m\left(\sup_{f\in F}|\mathrm{er}_P(f)-\widehat{\mathrm{er}}_{\mathbf{z}}(f)|\geqslant\varepsilon\right) \leqslant 4G^F(2m)e^{-m\varepsilon^2/8} \tag{7.27}$$

$$\leqslant 4e^{h\left(1+\ln\frac{2m}{h}\right)-m\varepsilon^2/8}. \tag{7.28}$$

Analogicznie do wzoru (7.21), powyższy rezultat można zapisać równoważnie w formie ograniczenia na błąd prawdziwy — z prawdopodobieństwem przynajmniej $1-\delta$ dla każdej funkcji $f \in F$ mamy

$$\mathrm{er}_P(f) \leqslant \widehat{\mathrm{er}}_{\mathbf{z}}(f) + \sqrt{\frac{h(1+\ln(2m/h))-\ln(\delta/4)}{m/8}}. \tag{7.29}$$

7.3.4 Funkcje rzeczywiste w uczeniu, pokrycia, 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 7.3.3 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 7.3.4 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

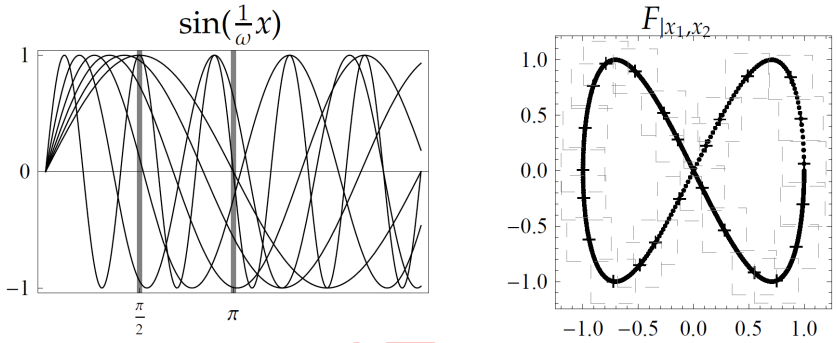
próby o danym rozmiarze m:

$$\mathcal{N}_d(\varepsilon, F, m) = \max\{\mathcal{N}\left(\varepsilon, F_{|\mathbf{x}_1,\ldots,\mathbf{x}_m}, d\right) : \ \mathbf{x} \in \mathbf{X}^m\}. \tag{7.30}$$

Należy zaznaczyć, że dla liczb pokryciowych używa się w ogólności metryk postaci

$$d_q(u,w) = \left(1/m \sum_i |u_i - v_i|^q\right)^{1/q}. \tag{7.31}$$

Z uwagi na czynnik $1/m$ zachodzi relacja: $\mathcal{N}_1(\cdot) \leqslant \mathcal{N}_2(\cdot) \leqslant \mathcal{N}_\infty(\cdot)$.

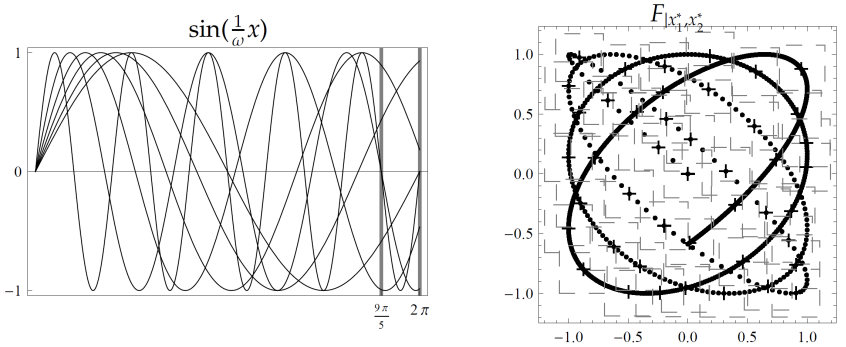


Rys. 7.8: Wykres kilku funkcji ze zbioru $F = \{\sin(\frac{1}{\omega}x) : \ \omega \in [0.2, 1]\}$ (lewa strona) oraz oraz przykładowe ε-pokrycie zbioru $F_{|\pi/2,\pi}$ — czyli pokrycie zamazania generowanego przez ten zbiór nad odciętymi $x_1 = \pi/2$, $x_2 = \pi$ (prawa strona). Pokrycie wyznaczone dla metryki d_∞ i $\varepsilon = 0.2$, zaznaczone na rysunku szarymi przerywanymi kwadratami (źródło: *opracowanie własne*).

Aby dać Czytelnikowi wizualne wyobrażenie o pojęciach ε-pokrycia, liczby pokryciowej i jednostajnej liczby pokryciowej przedstawiamy rysunki 7.8–7.11, na których rozważany jest następujący zbiór funkcji trygonometrycznych

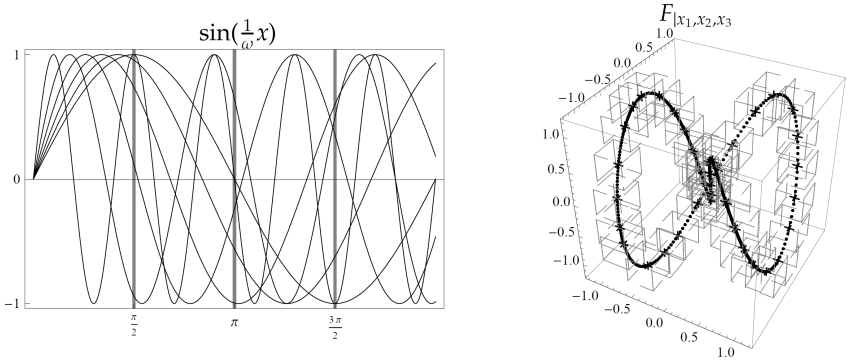
$$F = \left\{\sin\left(\frac{1}{\omega}x\right) : \ \omega \in [0.2, 1]\right\}, \tag{7.32}$$

z jednym parametrem ω decydującym o częstotliwości. Przyjęta jest metryka d_∞. Rys. 7.8 ilustruje przykładowe pokrycie zamazania generowanego przez zbiór F odciętego do dwuelementowej próby $x_1 = \pi/2$, $x_2 = \pi$. Rys. 7.9 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 7.10, 7.11 pokazują analogiczne przykłady dla prób trzyelelementowych.



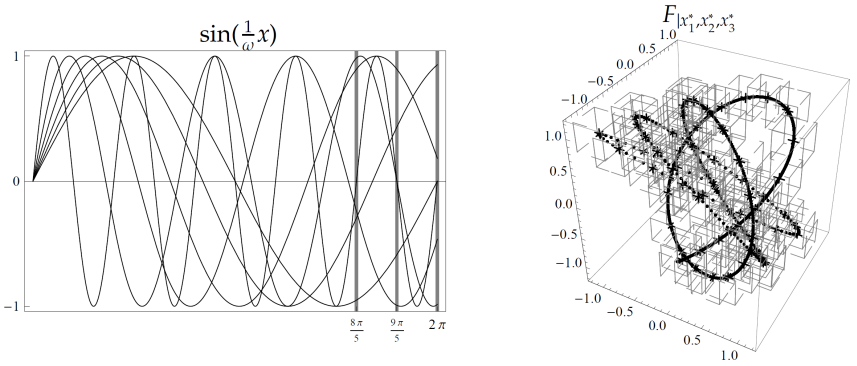
$$\mathcal{N}_\infty(0.2, F, 2) \approx 44$$

Rys. 7.9: 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*).



Rys. 7.10: 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 $F_{|\pi/2, \pi, 3/2\pi}$. Pokrycie wyznaczone dla metryki d_∞ i $\varepsilon = 0.2$, zaznaczone na rysunku szarymi przerywanymi sześcianami (źródło: *opracowanie własne*).

Istnieją następujące ważne twierdzenia o zbieżności jednostajnej wykorzystujące liczby pokryciowe.



$$\mathcal{N}_\infty(0.2, F, 3) \approx 66$$

Rys. 7.11: 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*).

Twierdzenie 7.3.2 (Anthony i Bartlett, [1, 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

$$P_m\left(\sup_{f\in F} \mathrm{er}_P(f) - \widehat{\mathrm{er}}^{\gamma}_{\mathbf{z}}(f) \geqslant \varepsilon\right) \leqslant 2\mathcal{N}_\infty(\gamma/2, F, 2m)e^{-\varepsilon^2 m/8}. \tag{7.33}$$

Twierdzenie 7.3.3 (Anthony i Bartlett, [1, 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

$$P_m\left(\sup_{f\in F} |\mathrm{er}_P(f) - \widehat{\mathrm{er}}_{\mathbf{z}}(f)| \geqslant \varepsilon\right) \leqslant 4\mathcal{N}_1(\varepsilon/16, F, 2m)e^{-\varepsilon^2 m/32}. \tag{7.34}$$

Twierdzenie 7.3.2 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.: $\mathrm{margin}(f(\mathbf{x}), y) = f(\mathbf{x}) - \frac{1}{2}$ dla $y = 1$ oraz

margin$(f(\mathbf{x}),y) = \frac{1}{2} - f(\mathbf{x})$ dla $y = 0$. Intuicyjnie: im większy margines, tym pewniejsze zaklasyfikowanie [2]. W twierdzeniu pojawia się $\widehat{\mathrm{er}}^{\gamma}_{\mathbf{z}}$, co oznacza częstość marginesu mniejszego niż γ na próbie, tj.: $\widehat{\mathrm{er}}^{\gamma}_{\mathbf{z}}(f) = \frac{1}{m}\sum_{i=1}^{m}[\mathrm{margin}\,(f(\mathbf{x}_i),y_i) < \gamma]$. 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 7.3.3 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 7.3.6) 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 7.3.4 (Haussler i Long, [22]) Niech F oznacza zbiór funkcji rzeczywistych $f\colon\ \mathbf{X} \to [0,1]$ o pseudowymiarze równym h. Wtedy:

$$\mathcal{N}_{\infty}(\varepsilon,F,m) \leqslant \sum_{i=0}^{h}\binom{m}{i}\lfloor 1/\varepsilon\rfloor^{i}, \tag{7.35}$$

co z kolei jest mniejsze niż $\left(\frac{me}{\varepsilon h}\right)^{h}$ dla $m \geqslant h$.

Twierdzenie 7.3.5 (Haussler, [21]) Niech F oznacza zbiór funkcji rzeczywistych $f\colon\ \mathbf{X} \to [0,1]$ o pseudowymiarze równym h. Wtedy:

$$\mathcal{N}_{1}(\varepsilon,F,m) \leqslant e(h+1)\left(\frac{2e}{\varepsilon}\right)^{h}. \tag{7.36}$$

Twierdzenie 7.3.6 (Zhang, [55]) 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 następująco: $\|\mathbf{x}_i\|_p \leqslant b$, $i = 1,\ldots,m$, gdzie $1/p + 1/q = 1$ (normy sprzężone) oraz $2 \leqslant p \leqslant \infty$.

[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 $\{0,1\}$.

Wtedy:

$$\mathcal{N}_2(\varepsilon, F, m) \leqslant (2n+1)^{\lceil a^2 b^2/\varepsilon^2 \rceil}. \tag{7.37}$$

Twierdzenie 7.3.6 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 $\ln \mathcal{N}_1(\cdot)$ oraz że $\mathcal{N}_1(\cdot) \leqslant \mathcal{N}_2(\cdot)$. I tak po zlogarytmowaniu (7.37) otrzymamy $\lceil a^2 b^2/\varepsilon^2 \rceil \ln(2n+1) = O(\ln n)$, zaś po zlogarytmowaniu (7.36) otrzymamy $(n+1)\ln(2e/\varepsilon) + \ln(n+2) + 1\ O(n)$, wiedząc, że wymiar VC wynosi $h = n+1$ dla funkcji liniowo zależnych od parametrów.

IV Systemy wnioskujące

8. Systemy produkcyjne i reprezentacja wiedz

TODO

9. Wnioskowanie

TODO

V Dodatki, spisy

10. Biblioteka *AISearch*

10.1 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 binarnym 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 {
  ...
}
```

10.2 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;
```

```
28
29      //utworzenie nowego id odpowiadającemu aktualnemu
            stanowi planszy
30      StringBuilder builder = new StringBuilder(parent.id);
31      builder[x*GridLength + y] = (char)(newValue + 48);
32      this.id = builder.ToString();
33
34      this.h = ComputeHeuristicGrade();
35  }
```

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:

```
1  ...
2  public class SudokuSearch : BestFirstSearch {
3      public SudokuSearch(SudokuState state) : base(state) { }
4      protected override void buildChildren(IState parent) {
5          SudokuState state = (SudokuState)parent;
6          //poszukiwanie wolnego pola
7          for (int i = 0; i < SudokuState.GRID_SIZE; ++i) {
8              for (int j = 0; j < SudokuState.GRID_SIZE; ++j){
9                  if (state.Table[i, j] == 0) {
10                     //wstawianie potomków w wolne pole
11                     for (int k = 1; k <
                           SudokuState.GRID_SIZE + 1; ++k) {
12                         SudokuState child = new
                               SudokuState(state, k, i, j);
13                         parent.Children.Add(child);
14                     }
15                     return;
16                 }
17             }
18         }
19     }
20     protected override bool isSolution(IState state) {
21         return state.H == 0.0;
22     }
23 }
```

Rozważmy sobie stan reprezentujący planszę sudoku, która ma następującą postać:

- 1 - 5 - 7 - 9 4	...	...
...	...	...
...	...	...

Jako stany potomne ww. planszy rozumiemy następujące plansze sudoku:

1 1 - 5 - 7 - 9 4	...	...
...	...	...
...	...	...

2 1 - 5 - 7 - 9 4	...	...
...	...	...
...	...	...

...

9 1 - 5 - 7 - 9 4	...	...
...	...	...
...	...	...

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();
        }
    }
```

10.3 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;
    }
```

```
16      //Dodanie stanu do potomków stanu rodzicielskiego
17      parent.Children.Add(this);
18  }
```

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:

		o	x		
		x	o		
	x	x	o		

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

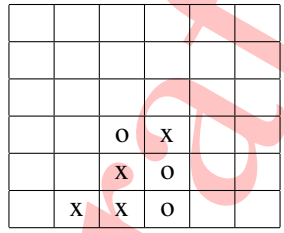
Nowo utworzoną pustą klasę należy zdefiniować jako publiczną i dziedziczącą po klasie bazowej oraz zaimplementować metody abstrakcyjne. W klasie wystarczy zdefiniować pusty konstruktor:

```
public Connect4Search(IState startState, bool
    isMaximizingPlayerFirst, int maximumDepth) :
    base(startState, isMaximizingPlayerFirst, maximumDepth)
    { }
```

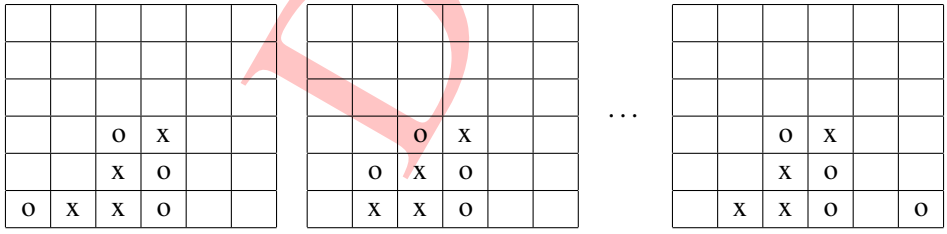
Parametry są następujące:

- **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

[1] M. Anthony i P.L. Bartlett. *Neural Network Learning: Theoretical Foundations*. Cambridge, UK: Cambridge University Press, 2009.

[2] 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.

[3] 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 (czer. 1998), s. 749–753. ISSN: 1057-7130. DOI: 10.1109/82.686696.

[4] 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.

[5] A. Brudno. "Bounds and valuations for shortening the search of estimates". W: *Problems of Cybernetics (Problemy Kibernetiki)* 10 (1963), s. 225–241.

[6] 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.

[7] Vladimir Cherkassky i Filip Mulier. *Learning from Data*. 2 wyd. USA: John Wiley & Sons, inc., 2007.

[8] 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.

[9] 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.

[10] 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}.

[11] D. Edwards i T. Hart. *The Alpha-Beta Heuristic*. Spraw. tech. 30. Massachusetts Institute of Technology, 1963.

[12] Neil Gershenfeld. *The Nature of Mathematical Modeling*. Cambridge, United Kingdom: Cambridge University Press, 1999.

[13] 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.

[14] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. 1st. USA: Addison-Wesley Longman Publishing Co., Inc., 1989. ISBN: 0201157675.

[15] T.D. Gwiazda. *Algorytmy genetyczne: Kompendium TOM 2*. Wydawnictwo Naukowe PWN, 2007. ISBN: 9788301153816.

[16] T.D. Gwiazda. *Algorytmy genetyczne: Kompendium TOM 1*. Wydawnictwo Naukowe PWN, 2009. ISBN: 9788301158309.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] John H. Holland. *Adaptation in Natural and Artificial Systems*. second edition, 1992. Ann Arbor, MI: University of Michigan Press, 1975.

[24] 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.

[25] Przemysław Klęsk. "Metoda nadawania pożądanych własności ekstrapolacyjnych neuronowym i rozmytym modelom systemów wielowymiarowych". praca doktorska. Politechnika Szczecińska, Wydział Informatyki, 2005.

[26] Przemysław Klęsk. "Zdolność do uogólniania w uczeniu maszynowym". praca habilitacyja. Zachodniopomorski Uniwersytet Technologiczny w Szczecinie, Wydział Informatyki, 2012.

[27] D.E. Knuth i R.W. Moore. "An analysis of alpha-beta pruning". W: *Artificial Intelligence* 6.4 (1975), s. 293–326.

[28] 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.

[29] R.E. Korf. "Depth-first Iterative-Deepening: An Optimal Admissible Tree Search". W: *Artificial Intelligence* 27 (1985), s. 97–109.

[30] Kevin P. Murphy. *Machine Learning: A Probabilistic Perspective*. The MIT Press, 2012. ISBN: 0262018020, 9780262018029.

[31] J. von Neumann. “Zur Theorie der Gesellschaftsspiele”. W: *Mathematische Annalen* 100.1 (1928), s. 295–320. DOI: 10.1007/BF01448847.

[32] J. von Neumann i O. Morgenstern. *Theory of Games and Economic Behavior*. Princeton University Press, 1944.

[33] A. Newell i A.H. Simon. “Computer Science and Empirical Inquiry: Symbols and Search”. W: *Commununications of the ACM* 19.3 (1976), s. 113–126.

[34] A.B. Novikoff. “On convergence proofs on perceptrons”. W: *Symposium on the Mathematical Theory of Automata*. Polytechnic Institute of Brooklyn, 1962.

[35] 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.

[36] Stanisław Osowski. *Sieci neuronowe do przetwarzania informacji*. I. Warszawa: Biuro Wydawnicze Politechniki of Warszawskiej, 2000. ISBN: 83–7207–187–X.

[37] 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.

[38] 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.

[39] Marcin Pietrzykowski. “Lokalne uczenie algorytmów regresyjnych metodą mini-modeli”. praca doktorska. Zachodniopomorski Uniwersytet Technologiczny w Szczecinie, 2019.

[40] Russel D. Reed i Robert J. Marks, II. *Neural Smithing*. London, England: A Bradford Book, The MIT Press, 1999. ISBN: 0–262–18190–8.

[41] 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.

[42] F. Rosenblatt. “The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain”. W: *Psychological Review* 65.6 (1958), s. 386–408.

[43] 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.

[44] S. Russell i P. Norvig. *Artificial Intelligence: A Modern Approach*. 3 wyd. Prentice Hall Press, 2009. ISBN: 978-0136042594.

[45] L. Rutkowski. *Metody i techniki sztucznej inteligencji*. Informatyka - Zastosowania. Wydawnictwo Naukowe PWN, 2012. ISBN: 9788301157319.

[46] N. Sauer. “On the density of families of sets”. W: *Journal of Combinatorial Theory, Series A* 13 (1972), s. 145–147.

[47] E. Shimon. *Graph Algorithms*. Cambridge University Press, 2011. ISBN: 978-0-521-73653-4.

[48] W. Shortz. *Sudoku Easy*. St. Martin’s Griffin, 2005.

[49] J.M. Steel. “Existence of submatrices with all possible columns”. W: *Journal of Combinatorial Theory, Series A* 24 (1978), s. 84–88.

[50] L.G. Valiant. “A theory of the learnable”. W: *Communications of the ACM* 27.11 (1984), s. 1134–1142.

[51] V.N. Vapnik. *The Nature of Statistical Learning Theory*. New York: Springer Verlag, 1995.

[52] V.N. Vapnik. *Statistical Learning Theory: Inference from Small Samples*. New York: Wiley, 1998.

[53] 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.

[54] D. Wolpert. “The Lack of A Priori Distinctions between Learning Algorithms”. W: *Neural Computation* 8 (1996), s. 1341–1390.

[55] Tong Zhang. “Covering Number Bounds of Certain Regularized Linear Function Classes”. W: *Journal of Machine Learning Research* 2 (2002), s. 527–550.

Źródła internetowe

[56] Wikipedia. *Sudoku*. Accessed: 22.12.2019. 2019. URL: http://en.wikipedia.org/wiki/Sudoku.

Spis rysunków

Spis tablic

Indeks

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

W

Z