

# Sztuczna inteligencja. Wykład wstępny

Paweł Rychlikowski

Instytut Informatyki UWr

24 lutego 2021

# O przedmiocie

(Ogólnym zasadom będzie poświęcone osobne spotkanie)

# Jaki język programowania?

Zapytanie: best computer language for ai

- Porada 1: Python, R, Lisp, Prolog, Java
- Porada 2: R, Python, Lisp, Java, Prolog
- Porada 3: Python, Java, Julia, Haskell, Lisp
- Porada 4: Python, C++, Lisp, Java, Prolog, Javascript, Haskell

Dlaczego tak (wybrane argumenty)?

- ① **Python**: prosty, uniwersalny, wiele bibliotek (m.in. uczenie maszynowe – ML), używany w wielu miejscach do nauki AI
- ② **R**: wsparcie dla ML
- ③ **C++, Java, Go, Rust, ...**: szybkość symulacji
- ④ **Lisp, Prolog**: dobre dopasowanie do **niektórych** zadań AI

- Stuart Russel, Peter Norvig, Artificial Intelligence. A Modern Approach. 3rd edition (w Internecie leży pdf)
- Fajna, ale 1100 stron.
- **Jest już czwarte wydanie!** (ale drogie)

Wykład na Stanfordzie

CS221: Artificial Intelligence: Principles and Techniques

## Definicja (krótka)

Zdolność komputera (programu) do wykonywania zadań powszechnie kojarzonych z zachowaniem intelligentnym (ludzi lub zwierząt).

## Definicja dłuża

Spis treści wybranego podręcznika o Sztucznej inteligencji

## Definicja długa – spis treści

Rozwiązywanie problemów przez przeszukiwanie, rozwiązywanie więzów, przeszukiwanie z oponentem, logika w opisie świata, wnioskowanie w logice, planowanie, modelowanie niepewności, reprezentacja wiedzy, wnioskowanie przy niepewności, podejmowanie decyzji, uczenie się z przykładów, uczenie się ze wzmacnieniem, przetwarzanie języka naturalnego, rozpoznawanie wzorców w obrazach i dźwiękach, robotyka, procedural content generation.

- 1.
- 2.
- 3.
- 4.
- 5.
- 6.

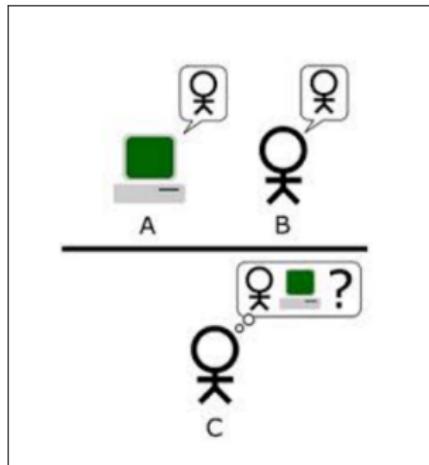
## Definicja długa – spis treści

Rozwiązywanie problemów przez przeszukiwanie, rozwiązywanie więzów, przeszukiwanie z oponentem, logika w opisie świata, wnioskowanie w logice, planowanie, modelowanie niepewności, reprezentacja wiedzy, wnioskowanie przy niepewności, podejmowanie decyzji, uczenie się z przykładów, uczenie się ze wzmacnieniem, przetwarzanie języka naturalnego, rozpoznawanie wzorców w obrazach i dźwiękach, robotyka, procedural content generation.

## Przedmioty:

1. Sztuczna inteligencja (nasz przedmiot)
2. Sieci neuronowe, Machine learning, Eksploracja danych
3. Przetwarzanie języka naturalnego, Eksploracja tekstów, ...
4. Model checking, Sztuczna inteligencja i logika, ...
5. Probabilistic graphical models
6. Sztuczna inteligencja w grach

# Test Turinga



- Pytanie: Czy maszyny potrafią myśleć?
- Odpowiedź: Możemy to sprawdzić w rozmowie.

## Pytanie

Czy naprawdę chcemy, żeby komputer był nieroóżnialny w rozmowie od człowieka?



źródło: [tvtropes.org](http://tvtropes.org)

Nieludzkie, pożądane cechy: wiedza, zdolności rachunkowe, modelowanie świata, rozwiązywanie zadań fizycznych, ...

- Przeszukiwanie (problem solving)
- Uczenie się (learning)
- Wnioskowanie (inference, theorem proving)
- Modelowanie wiedzy o świecie

## Główny problem SI

Głównym, nierozwiążanym do tej pory problemem SI jest satysfakcyjujące połączenie tych światów, zwłaszcza wnioskowania (twarda logika), z rozmytą wiedzą pochodząą z uczenia się.

Podajemy początkowy fragment wypowiedzi, następnie komputer wymyśla *dalszy ciąg*. Jest to najbardziej zaawansowany system **generowania tekstu**, czyli zadania ze świata *rozmytego*

A gdyby próbować nakłonić ten system, by zaprezentował jakieś rozumowanie?

**Suppose I have 4 apples, and I gave two of them to my sister,** and she gave one of them to her sister, and it started to grow. Would I be able to see how much time it had taken for the apple to grow?

We will first give you a number which you will later use to work out how much time it has taken for the apple to grow:

(Time it took to grow 2 apples, shown as a clock)  $\times$  (Size of apple)  
= "time taken"

What this tells us is that:

$$2 \times 3 \times 2 = 44 \text{ (seconds in 24 hours)}$$

So the question is, what does 44 seconds look like?

Let's do a quick experimen

# Dlaczego problemy SI są trudne

Istnieją dwa główne powody

① Trudność algorytmiczna:

- **NP-zupełne** - łamigłówki,
- **PSPACE-zupełne** - gry,
- **EXPTIME i wyżej** – systemy wnioskowania

② Czasem trudno sformalizować precyzyjnie problem!

## Przykład

Rozważmy zadanie generowania tekstów w języku naturalnym (polskim, angielskim). Co to znaczy, że komputer robi to dobrze? Czy [talktotransformer.com](https://talktotransformer.com) robi to dobrze?

# Co się udało Sztucznej inteligencji?

Gdzie komputery są od nas lepsze?

- ➊ Oczywiście my też wykonujemy algorytmy (np. mnożenie liczb). Tu od zawsze **one** mają przewagę.
- ➋ Komputery radzą sobie w grach: Szachy (1996), Go (2016), Poker (2017), StarCraft (AlphaStar, 2019?)
- ➌ Rozpoznawanie mowy (Microsoft/IBM, Switchboard Corpora, około 2016)

## Gdzie komputery są od nas lepsze?

- ① Rozpoznawanie prostych obrazów (ImageNet, 1mln obrazków, 1000 klas):
  - 2010: około 28% błędów
  - 2015 (4.94% na ImageNet, człowiek: 5.1%)
  - Obecny rekord: 1.2% (Top5 accuracy, 2021), w 2020 było 1.3%
- ② Tłumaczenie maszynowe (może niekoniecznie najbardziej skomplikowanych tekstów)
- ③ Wygrywanie teleturniejów wiedzowych (Watson, Jeopardy, 2011)

# Co działa niekoniecznie idealnie?

- ① Rozpoznawanie mowy (50% błędów na nagraniach *przy kotlecie*)
- ② Roboty umiejscowione nalewać wodę, otwierać drzwi, itd w **nieznanym** środowisku
  - DARPA challenge fails
- ③ Gra w Brydża
- ④ „Ludzka” rozmowa na dowolny temat

# Paradoks Morav(e)ca

Paradoks, ok. 1980

Stosunkowo łatwo sprawić, żeby komputery przejawiały umiejętności dorosłego człowieka w testach na inteligencję albo w grze w warcaby, ale jest trudne albo wręcz niemożliwe zaprogramowanie im umiejętności rocznego dziecka w percepji i mobilności. [Zgadzamy się?](#)

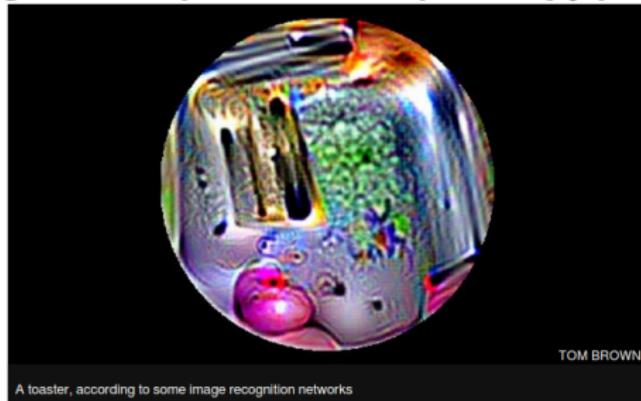
Steven Pinker

Gdy pojawi się nowa generacja inteligentnych urządzeń, to analitycy giełdowi, inżynierowie i ławnicy sądowi mogą zostać zastąpieni maszynami. Ogrodnicy, recepcjonisci i kucharze są bezpieczni w najbliższych dekadach

Łatwiej nam programować to co świadome (bo to lepiej rozumiemy), niż nieświadomość.

# Rozpoznawanie obrazów. Super toster

Sztucznie wygenerowany obraz, maksymalizujący **tosterowatość**.



# Co możemy zrobić z tym obrazkiem?

- Możemy go pokazywać sieci.
- Ale wklejając go analogowo, nie cyfrowo.
- Zobacz pracę: Adversarial Patch, T. Brown i inni, 2017

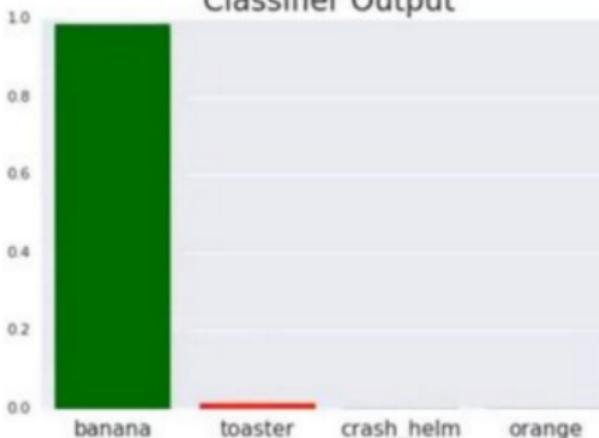


# Tostery i banany

Classifier Input



Classifier Output



# Tostery i banany

Classifier Input



Classifier Output



# Tesla i znaki ograniczenia szybkości

Ograniczenie do 85 mil na godzinę



Najpierw zajmiemy się **przeszukiwaniem**, które jest podstawowym narzędziem AI.

Koniec części I

# Problem solving by searching. Intuicje

## Przykład 1. Wyznaczanie trasy



## Przykład 2. Wyznaczanie sekwencji działań

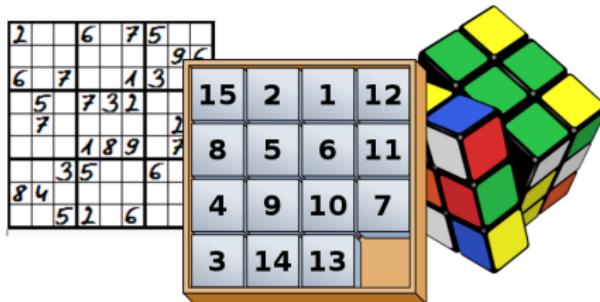
Kohler (1945): monkey and banana problem.



Kohler observed that chimpanzees appeared to have an insight into the problem before solving it

# Problem solving by searching. Intuicje

## Przykład 3. Rozwiązywanie łamigłówek



### Uwaga

Problemy zabawkowe (toy problems) są częstym narzędziem w AI.

## Definicja

**Problem** ma następujące pięć komponentów

1. Stan początkowy (i zbiór stanów, ale być może dany implicitznie)
2. Zbiór akcji (co **agent** może robić)
3. Model przejścia (**stan + akcja = nowy\_stan**)
4. Test określający, czy stan jest końcowy (i znaleźliśmy rozwiązanie)
5. Sposób obliczania kosztu ścieżki (najczęściej podawany jako koszt akcji w stanie)

**Agent** – program, który odbiera wrażenia o świecie, podejmuje decyzje, wykonuje akcje, maksymalizując jakąś funkcję celu.

## Uwaga

Akcje oraz model przejścia są ze sobą powiązane. Jest kilka wariantów:

- ① Zbiór akcji wspólny, funkcja przejścia pilnuje, żeby **niemożliwe** akcje nie zmieniały stanu  
*Przykład: ludzik idzie w labiryncie na ścianę*
- ② Akcje to funkcja, która dla stanu zwraca zbiór czynności, które agent może zrobić w stanie
- ③ Akcja i model przejścia to jedna funkcja, która dla stanu zwraca **listę** par postaci:  
(akcja, nowy\_stan)

# Poziomy abstrakcji.

Zadania można modelować na wiele różnych sposobów. Dla podróżowania MPK mamy następujące akcje/sekwencje akcji

- Przejedź z przystanku A do przystanku B
- Wsiądź do tramwaju na przystanku A, skasuj bilet, zajmij wygodne miejsce, obserwuj tablicę, podejdź do drzwi, gdy...
- Wykonuj naprzemienne ruchy lewą i prawą nogą, aż ...
- Napnij głowę większą mięśnia dwugłowego lewego uda, ...

## Uwaga

Akcje muszą być zrozumiałe dla agenta, im wyższy poziom, tym łatwiejsze zadanie przeszukiwania.

- Przeanalizujemy dla kilku przykładów jak można pewne zadania przedstawiać jako problemy przeszukiwania

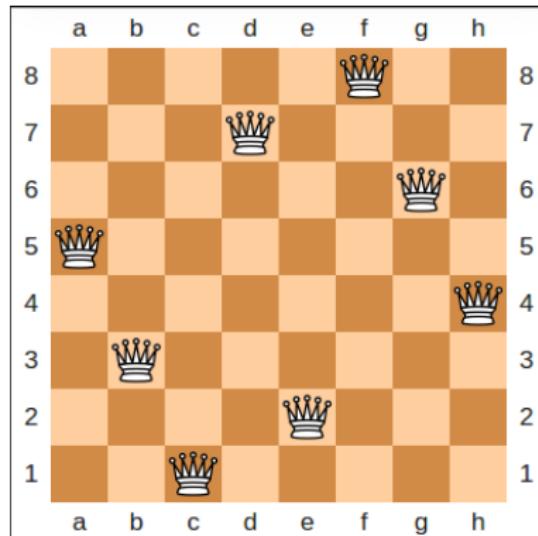
# Piętnastka



- **Stan początkowy:** jakieś ułożenie, na przykład powyższe
- **Stan końcowy:** liczby po kolej
- **Koszt:** jednostkowy
- **Model:** dowolna zamiana pustego kwadracika z sąsiadem

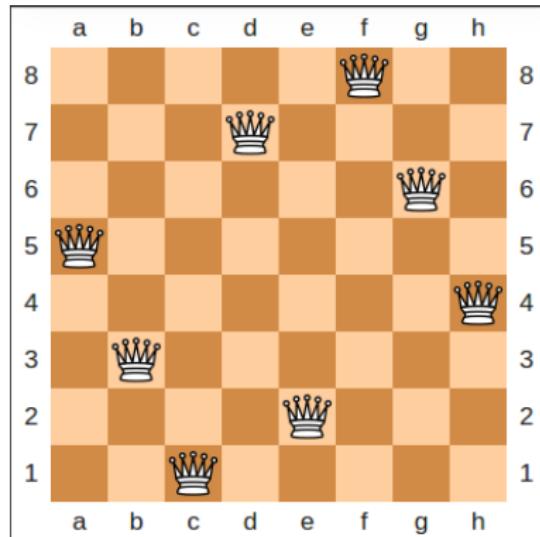
- **Stan początkowy:** jakieś ułożenie dwóch wież i królów, informacja o tym, kto się rusza
- **Stan końcowy:** mat
- **Koszt:** jednostkowy
- **Model:** ruch szachowy + zmiana gracza aktywnego

# Problem ośmiu hetmanów



- W przeciwieństwie do poprzedniego przykładu istnieje tu wiele sformułowań.
- Jakie są dwie najważniejsze opcje?

# Problem ośmiu hetmanów



Dwa sposoby opisywania zadania jako problemu przeszukiwania:

- ① **Stan kompletny** (rozwijaćmy sytuacje z 8 hetmanami na planszy, ruch to przestawienie hetmana)
- ② **Stan niepełny** – zaczynamy od pustej planszy, ruchem jest postawienie hetmana.

## Uwaga

Rozmieszczamy hetmany wg określonego schematu, rozpoczynając od pustej planszy. Liczymy liczbę stanów.

- $64 \times 63 \times \dots \times 57 = 178462987637760$  – rozmieszczamy po kolejni hetmany, na jednym polu jest 1 hetman.
- $8^8 = 16777216$  – ustalona kolejność, w każdej kolumnie 1 hetman
- $8! = 40320$  – ustalona kolejność, w każdej kolumnie 1 hetman, nie szachują się w wierszach

# Problem zabawkowy: hipoteza Knutha

## Hipoteza

Zaczynając od 4 możemy dojść do dowolnej liczby wykonując operacje: silni, pierwiastka i podłogi (części całkowitej, int).

Przykładowo:

$$\lfloor \sqrt{\sqrt{\sqrt{\sqrt{(4!)!}}}} \rfloor = 5$$

Sformułowanie dość oczywiste (stany to liczby)

## Uwaga

Przestrzeń jest nieograniczona (nie znamy żadnego twierdzenia, które ograniczałoby przestrzeń prostą funkcją).

# Modyfikacje zadania znajdowania marszruty

Przesuwamy się w stronę rzeczywistych zadań.

## Wariant 1

W co  $K$ -tej miejscowości na trasie znajduje się dobra knajpa.

## Wariant 2

Powinniśmy zaliczyć co najmniej  $K$  dodatkowych atrakcji turystycznych.

Zastanówmy się, co jest stanem w powyższych zadaniach?

## Wariant 1

W co  $K$ -tej miejscowości na trasie znajduje się dobra knajpa.

Stan: (miejscowość, licznik-modulo- $K$ )

## Wariant 2

Powinniśmy zaliczyć co najmniej  $K$  dodatkowych atrakcji turystycznych.

Stan (1): (miejscowość, liczba-atrakcji-do-zaliczenia) **źle!**

Stan (2): (miejscowość, zbiór-zaliczonych-atrakcji) **lepiej!**

Przykładowe zadania z rzeczywistego świata, będące zadaniami przeszukiwania:

- Różne zagadnienia logistyczne
- Projektowanie układów scalonych
- Nawigacja robotów
- Organizacja procesu produkcji
- Tworzenie projektów o określonych właściwościach

# Sztuczna inteligencja. O przeszukiwaniu

Paweł Rychlikowski

Instytut Informatyki UWr

3 marca 2021

## Zadanie o Panu Tadeuszu

- a) **Przeszukujemy** możliwe podziały (sposoby legalnego wstawiania spacji)
- b) Można myśleć o tym jako o **modelu** języka: język składa się raczej z długich, niż z krótkich słów
- c) Uwaga: konieczny jest algorytm dynamiczny!

## Zadanie o Pokerze

- a) Najpierw zagadka: Jaki jest związek tego zadania z pasjansem i bombą atomową?

# Rozwiążanie zagadki

- Stanisław Ulam pracował w projekcie Manhattan
- Kiedyś, w czasie rekonwalescencji po operacji układął pasjanse. A te wiadomo, czasem wychodzą, czasem nie.
- Chciał obliczyć prawdopodobieństwo (że pasjans wyjdzie), ale mu nie wychodziło

Wymyślił, że można je obliczyć symulując wiele rozdań na komputerze! (w roku 1947)

Tak powstały metody Monte Carlo (do wykorzystania w wielu miejscach, między innymi w zadaniu o pokerze).

## Fragment treści zadania

Sprawdź za pomocą tego programu (wykonując kilka eksperymentów), jak zmienia się prawdopodobieństwo sukcesu, jeżeli pozwolimy Blotkarzowi wyrzucić pewną liczbę wybranych kart przed losowaniem (inaczej mówiąc, pozwalamy Blotkarzowi na skomponowanie własnej talii, oczywiście złożonej z blotek).

*Czy potrafisz skomponować zwycięską talię dla Blotkarza (mającą możliwie dużo kart)?*

- **ograniczenie:**  $p_{\text{Blotkarz wygrywa}} > \frac{1}{2}$ ,  
**cel optymalizacji:** maksymalna liczba kart
- Inna możliwość: wspólne kryterium liczbowe (wiele możliwości)
- Optymalizacja wielokryterialna (Optimum w sensie Pareta, będzie zadanie na liście C1)

# Oczekiwania wobec rozwiązania

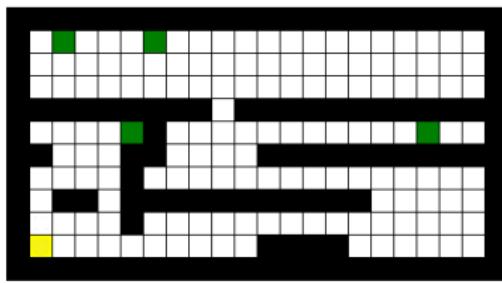
(wracamy do ogólniejszych zagadnień)

- **Zupełność:** czy program znajdzie drogę do rozwiązania, jeżeli takowa istnieje?  
*Czy to jest konieczny warunek użyteczności algorytmu?*
- **Optymalność:** czy będzie ona najkrótsza
- **Złożoność czasowa:** jak długo będzie trwało szukanie
- **Złożoność pamięciowa:** ile zużyjemy pamięci

## Pytanie

Dlaczego tak ważna jest złożoność pamięciowa?

- Agent **bez pamięci** z konieczności operuje drzewem przeszukiwań (bo nie potrafi stwierdzić, że w jakimś stanie już był)
- Najczęściej lepiej modelować świat za pomocą **grafu**



- Będziemy teraz rozważać różne labirynty, na kwadratowej siatce.
- Labirynt jako problem wyszukiwania:
  - **stan** – współrzędne pola na którym można staćć (nie ściany)
  - **start** – ustalona pozycja w labiryncie (**żółta**)
  - **cel** – ustalone pozycje w labiryncie (**zielona**)
  - **model** – 4-sąsiedztwo (modulo ściany), akcje to **N, W, E, S.**
  - **koszt** – jednostkowy

Można wzbogacić przestrzeń stanów w labiryncie

- Dodać drzwi i klucze (czym stanie się **stan**)?
- Dodać poruszających się (deterministycznie) wrogów (**stan?**)
- Dodać skrzynie z bronią, apteczki i punkty życia (**stan?**)

Koniec nagrania I

**BFS** = Breadth First Search

## Opis

- Mamy 3 grupy stanów: **do-zbadania**, **zbadane** i pozostałe.
- Na początku mamy 1 stan **do-zbadania**: stan startowy
- Stany do zbadania przechowujemy w kolejce **FIFO** (first-in first out)
- Badanie stanu:
  - Sprawdzenie, czy jest stanem docelowym (jak tak, to **koniec!**)
  - Ustalenie, jakie akcje możemy zrobić w tym stanie, znalezienie nowych stanów **do-zbadania**

## Skrócony opis

Pobieraj stan z **kolejki**, przetwarzaj, jak kolejka się skończy (nic **do-zbadania**) to zakończ działanie, (możesz też zakończyć, jak znajdziesz stan docelowy).



**DFS** = Depth First Search

## Opis

- Stany przetwarzamy w innej kolejności: dzieci aktualnie rozwijanego mają priorytet
- Czyli zamiast FIFO używamy LIFO (List in First out), czyli po prostu stosu.
- Oprócz tego algorytm się nie zmienia.

**DLS** = Depth Limited Search

## Opis

- Określamy maksymalną głębokość poszukiwania.
- Przeszukujemy w głąb, ale nie rozwijamy węzłów na głębokości większej niż  $L$ .
- Wygodnie implementuje się rekurencyjnie (proste ćwiczenie)

- W algorytmach na grafach używa się takich parametrów jak  $|V|$  oraz  $|E|$  (liczba stanów, liczba krawędzi)
- Dobra złożoność to może być  $O(|V| + |E|)$
- W sztucznej inteligencji, gdzie często nie znamy grafu (lub jest on zbyt duży, żeby traktować go jako daną do zadania), używamy innych parametrów

## Parametry zadania wyszukiwania

- $b$  – maksymalne rozgałęzienie (branching factor)
- $d$  – głębokość najgłębszego węzła docelowego
- $m$  – maksymalna długość ścieżki w przestrzeni poszukiwań

## Uwaga

Mówiąc o czasie (pamięci) często używamy jako jednostki liczby węzłów (przetworzonych/pamiętanych).

# Czas i pamięć dla BFS i DFS

## BFS

Czas =  $O(b + b^2 + b^3 + \dots + b^d) = O(b^d)$

Pamięć = Czas

**Uwaga:** Może być też  $O(b^{d+1})$  jak testujemy warunek sukcesu dopiero podczas rozwijania.

## DFS

Czas =  $O(b^m)$  – niedobrze

Pamięć =  $O(bm)$  – dobrze

## Uwaga

W tych rozważaniach zakładamy, że przestrzeń jest tak wielka, że nie spamiętujemy odwiedzonych stanów (względnie wiemy, że stany się nie powtarzają)

Oczywiście w skończonych grafach nasze rozważania są nazbyt pesymistyczne!



## Uwaga

**Iteracyjne pogłębianie** to po prostu wywoływanie DLS na coraz to większej głębokości (bez zapamiętywania żadnych pośrednich wyników)

Może wydawać się to stratą czasu, ale:

- działaemy w pamięci  $O(bd)$ ,
- na czas wpływa ostatnia warstwa, czyli  $O(b^d)$

- UCS = Uniform Costs Search
- Zamiast kolejki FIFO mamy kolejkę priorytetową, z priorytetem równym kosztowi dotarcia do węzła.

## Uwaga

Oczywiście umożliwia to różnicowanie kosztów dotarcia z węzła do węzła.

## Uwaga 2

UCS rozwiązuje ten sam problem co algorytm Dijkstry (i w bardzo podobny sposób). Ale jest różnica powiedzmy **filozoficzna**

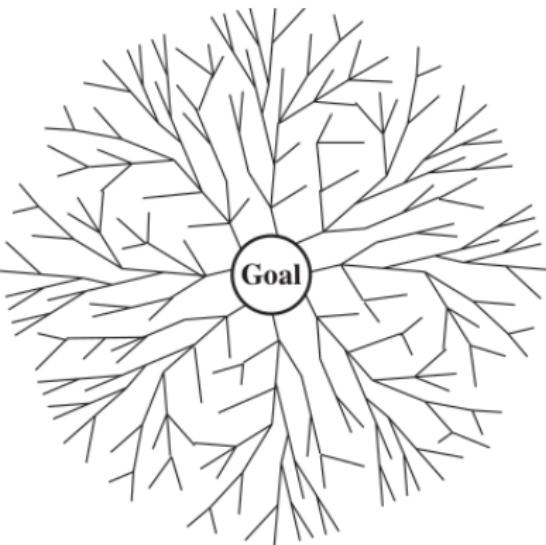
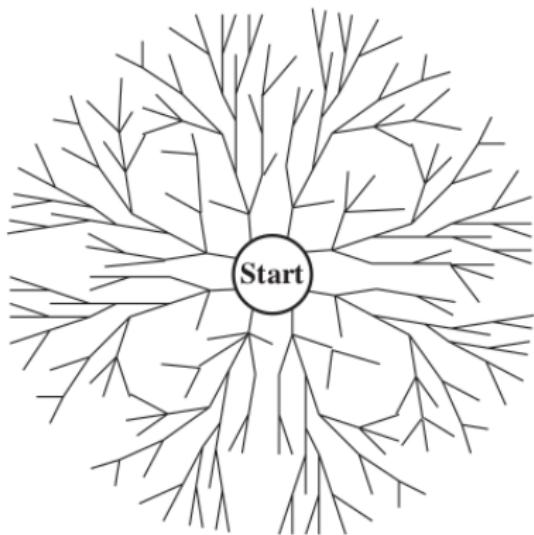
# Uniform Cost Search a Dijkstra

- UCS jest na sztucznej inteligencji, Dijkstra na algorytmach (to oczywiście nie jest poważna różnica).
- UCS jest przedstawiany najczęściej jako instancja algorytmu typu **Best First Search**
- Graf który przeszukujemy może być duży, nieznany w całości, nieskończony, itd.

## Pomysł

Prowadźmy poszukiwania jednocześnie od przodu i od tyłu

Rysunek:



## Problemy

Nie zawsze jest możliwe do zastosowania:

- ① Musimy znać stan końcowy (vide hetmany czy obrazki logiczne)
- ② Najlepiej jak jest jeden (albo niewiele i umiemy je wszystkie wymienić)
- ③ Musimy umieć odwrócić funkcję następnika (vide problem Knutha i funkcja `int ( )`)
- ④ Musimy pamiętać odwiedzone stany (przynajmniej z jednej strony)  
**BFS + IDS** (lub **BFS + BFS**) zamiast **IDS+IDS**

## Korzyści

Podstawowa korzyść to czas działania. Dlaczego?

Odpowiedź: Zamiast jednego przeszukania na głębokości  $d$  mamy dwa przeszukania na głębokości  $d/2$ .

# Przeszukiwanie bez wiedzy. Podsumowanie

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes <sup>a</sup>	Yes <sup>a,b</sup>	No	No	Yes <sup>a</sup>	Yes <sup>a,d</sup>
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes <sup>c</sup>	Yes	No	No	Yes <sup>c</sup>	Yes <sup>c,d</sup>

**Figure 3.21** Evaluation of tree-search strategies.  $b$  is the branching factor;  $d$  is the depth of the shallowest solution;  $m$  is the maximum depth of the search tree;  $\ell$  is the depth limit. Superscript caveats are as follows: <sup>a</sup> complete if  $b$  is finite; <sup>b</sup> complete if step costs  $\geq \epsilon$  for positive  $\epsilon$ ; <sup>c</sup> optimal if step costs are all identical; <sup>d</sup> if both directions use breadth-first search.

Koniec nagrania II

# Problemy bezczujnikowe (sensorless)

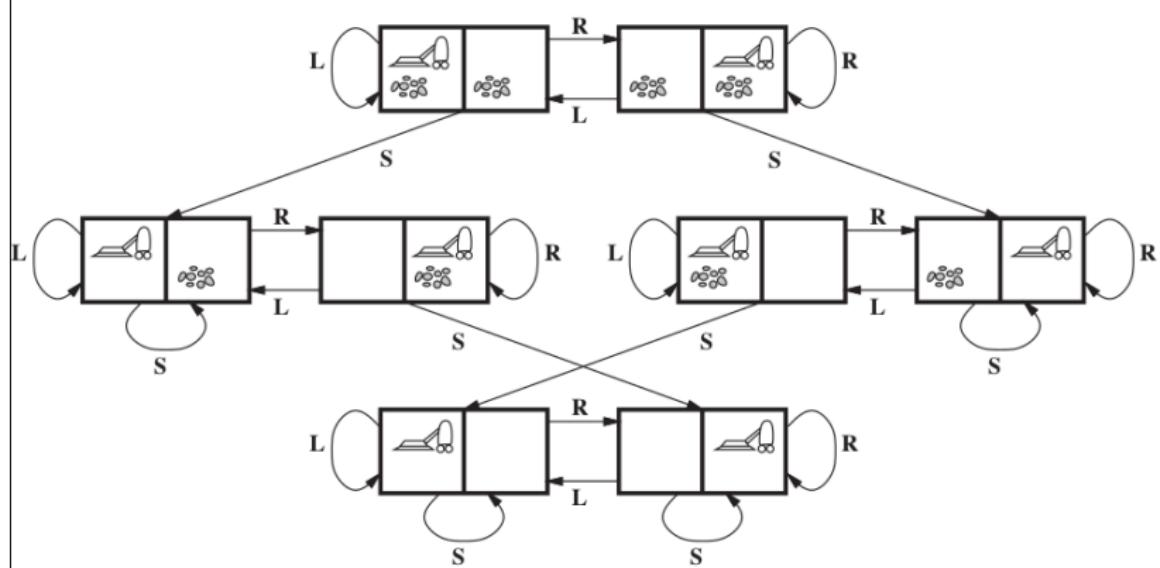
- Czujniki są drogie. Czasem wolimy na przykład znaleźć sekwencje akcji, która doprowadzi do celu niezależnie od stanu.
- Przykład 1 Szeroko działający antybiotyk
- Przykład 2 Robot w linii produkcyjnej, który składa jakieś części wykonując akcje niezależne od tego, jak te części się ułożyły.

## Uwaga

Oczywiście rozwiązywanie problemu bezczujkowego nie jest optymalne w środowisku z dostępem do sensorów. Zakładamy na przykład, że pewne akcje będą „puste”.

# Problemy bezczujnikowe (przykładowy odkurzacz)

Wszyscy wiemy o inteligentnych odkurzacach. Ten będzie trochę prostszy (rysunek z przestrzenią stanów):



## Definicja

**Stanem przekonań** jest zbiór **stanów** oryginalnego problemu, w których agent (być może) się znajduje.

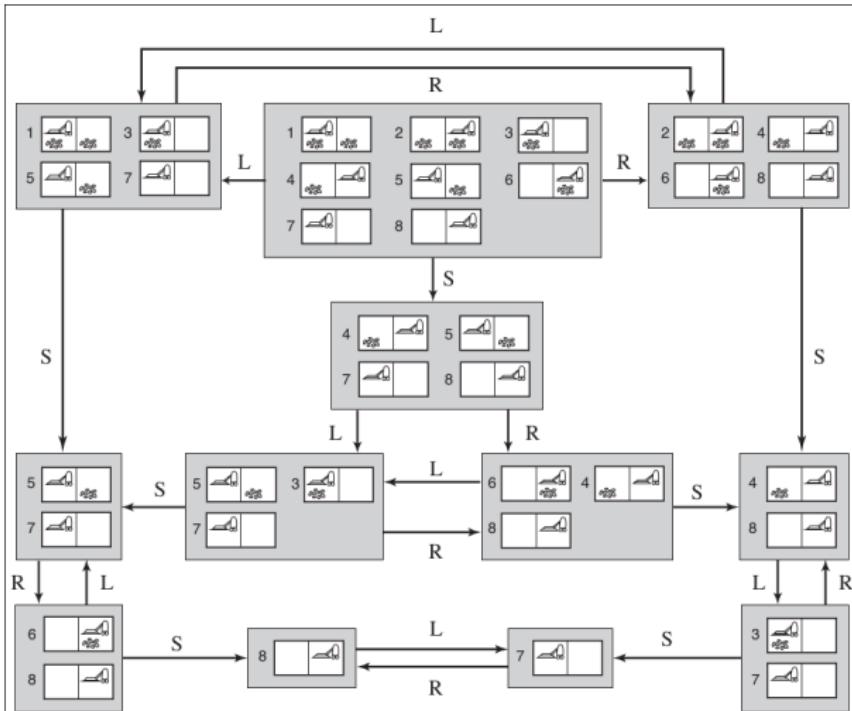
## Pytanie 1

Jak się poruszać w takiej przestrzeni?

## Pytanie 2

Jaka sekwencja akcji jest rozwiązaniem problemu bezczujnikowego (napiszmy ją na tablicy).

# Przestrzeń przekonań odkurzacza. Przykład



(pętle dla wszystkich stanów usunięte ze względu na czytelność.)

- 1 Przejścia w **przestrzeni przekonań** powstają przez zaaplikowanie funkcji przejścia do **stanu** (obliczenia obrazu funkcji)
- 2 **Stan** jest końcowy jeżeli wszystkie **stany** w nim zawarte są końcowe.
- 3 Koszt jednostkowy (spory problem w innym przypadku)
- 4 **Stan startowy**: zbiór wszyskich **stanów**.

# Komandos z mapą. Mniej trywialny przykład

- Rozważmy zadanie, w którym do labiryntu wrzucony zostaje komandos z mapą...
- ale zrzut jest w nocy i nie wiadomo, gdzie trafił.
- Problem:  
*znajdź sekwencję akcji, która **na pewno** doprowadzi do jednego z celów (akcje niedozwolone nie przesuwają komandosa).*

- Zadanie z komandosem będzie na liście P2.
- Zbadajmy, jak działa taka przestrzeń przekonań.

## Zmniejszanie niepewności

Zobaczmy, jakie są możliwości **zmniejszania niepewności** w tym zadaniu (program `commando_z_wykladu.py`).

- Opłaca się iść **w kierunku** rozwiązania.
- Co to oznacza?

Zakładamy, że umiemy **szacować** odległość od rozwiązania.

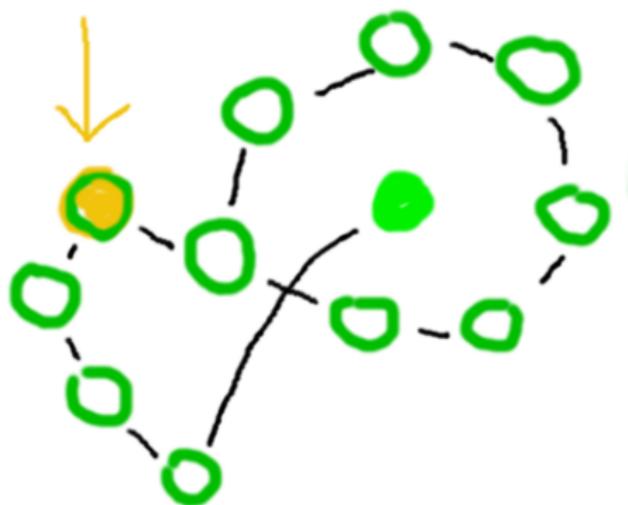
## Przykłady

- 1 Odległość w linii prostej w zadaniu szukania drogi.
- 2 Odległość taksówkowa (Manhattan distance) w labiryncie.

- Rozwijamy ten węzeł, który wydaje się najbliższego rozwiązania.
- Proste, intuicyjne, ale są problemy. Jakie?

Można ten algorytm „oszukiwać”, w skrajnym przypadku sprawić, żeby rozwiązanie w ogóle nie zostało znalezione (w wersji bez zapamiętywania stanów, w których byliśmy).

# Plansza nieprzyjazna dla algorytmu zachłannego



# Sztuczna inteligencja. Przeszukiwanie z wiedzą o problemie

Paweł Rychlikowski

Instytut Informatyki UWr

10 marca 2021

# Algorytm A\*

## Definicje

- $g(n)$  – koszt dotarcia do węzła  $n$
- $h(n)$  – szacowany koszt dotarcia od  $n$  do (najbliższego) punktu docelowego ( $h(s) \geq 0$ )
- $f(n) = g(n) + h(n)$

## Algorytm

Przeprowadź przeszukanie, wykorzystując  $f(n)$  jako priorytet węzła (czyli rozwijamy węzły od tego, który ma najmniejszy  $f$ ).

# Algorytm A\*. Uwagi

- Zwróćmy uwagę, że algorytm przypomina BFS (w którym, jak pamiętamy, używamy kolejki FIFO) oraz algorytm UCS (uniform cost search, Dijkstry).
- Jedyną różnicą między A\* i UCS jest użycie funkcji  $f$ , a nie funkcji  $g$  jako priorytetu w kolejce priorytetowej.

Oczywiście od wyboru funkcji **h** (nazywanej **heurystyką**) zależą właściwości algorytmu  $A^*$

Wymienimy najważniejsze właściwości funkcji  $h$ .

1. **Nieujemna**:  $h(s) \geq 0$ , dla każdego  $s$

2. **Rozsądna**:  $h(s_{\text{end}}) = 0$

3. **Dopuszczalna** (admissible):

$h(s) <$  prawdziwy koszt dotarcia ze stanu  $s$  do stanu końcowego

Inaczej: **optimistyczna**

4. **Spójna** (consistent),  $s_1, s_2$  to sąsiednie stany:

$$h(s_2) + \text{cost}(s_1, s_2) \geq h(s_1)$$

Ostatnia własność przypomina **własność trójkąta** w definicji metryki.

Popatrzmy na rysunek

# O optymizmie

Pojęcie **optymistyczna** wydaje się dość intuicyjne w kontekście heurystyki.

Zwróćmy uwagę, że:

- Dla zadania: dojechać z punktu  $A$  do  $B$  po drogach publicznych, heurystyka szacująca koszt dotarcia do  $B$  jako odległość euklidesową z punktu  $X$ , w którym jesteśmy, do celu jest optymistyczna:  
zakładamy bowiem optymistycznie, że istnieje prosta, pozbawiona zakrętów droga prościutko do  $B$
- Jeżeli podróżujemy po Manhattanie (czyli w miejscu, gdzie wszystkie drogi przecinają się pod kontem prostym), poprzednia heurystyka nadal będzie optymistyczna. Ale bardziej realistyczne będzie liczenie tzw. odległości taksówkowej, która jest po prostu sumą różnic na współrzędnych  $x$  oraz  $y$ .

(zobaczmy rysunek)

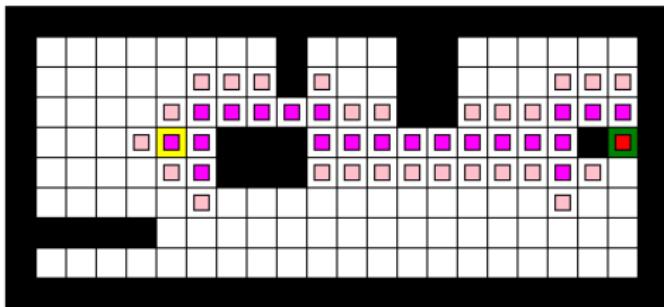
# O optymizmie

Jak za chwilę zobaczymy, wybór bardziej realistycznej (ale ciągle optymistycznej) heurystyki zaowocuje lepszym działaniem algorytmu A\*.

- Ponieważ  $h$  ma być szacowaną odległością od celu, umówimy się, że własność **nieujemności i rozsądności** funkcji  $h$  są konieczne, żeby mówić o algorytmie A\*
- Pozostałe dwie własności z poprzedniego slajdu (optymistyczna i spójna) są „mile widziane”, ale niekonieczne.

- ① UCS to  $A^*$  z super-optymistyczną heurystyką ( $h(s) = 0$ )
- ② Spójna heurystyka jest optymistyczna  
Dowód: Indukcja po węzłach (na ćwiczeniach, okolice C2)

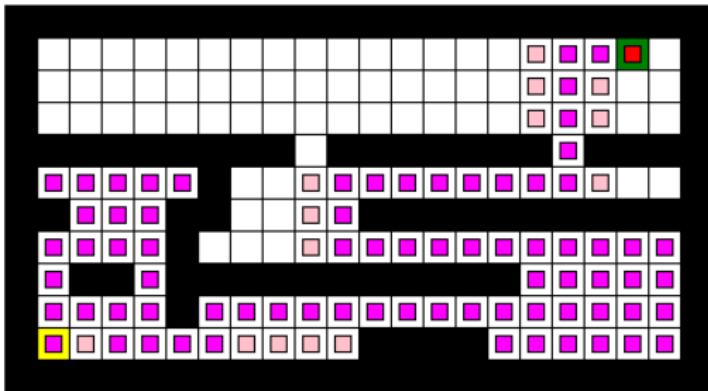
# $A^*$ w labiryncie (1)



Używamy odległości taksówkowej między bieżącą kratką a celem jako heurystyki (czyli **optymistycznie** zakładamy, że nie spotkamy po drodze żadnej ściany).

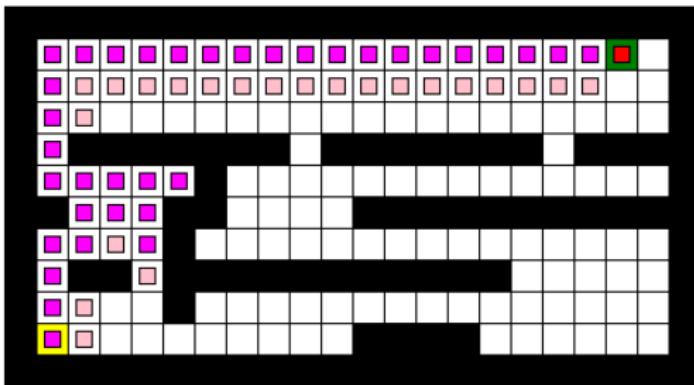
Kolor **różowy**: węzy w kolejce, kolor **purpurowy** – węzły rozwinięte. Jedynie dwa rozwinięte węzły są poza optymalną ścieżką. Na tym rysunku (i kolejnych) widzimy stan algorytmu w momencie osiągnięcia węzła końcowego.

## $A^*$ w labiryncie (2)



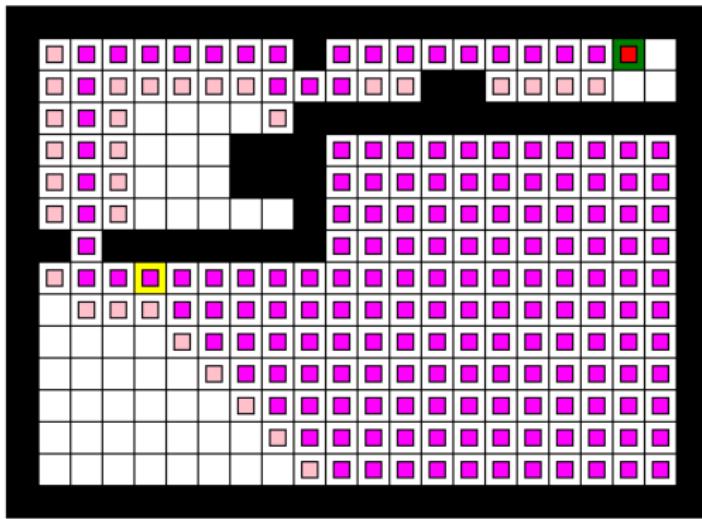
W dolnej części labiryntu heurystyka trochę prowadzi na manowce

# $A^*$ w labiryncie (3)



ale jeżeli w poprzednim labiryncie przebić drzwi, to wówczas znowu jest prawie idealnie.

# $A^*$ w labiryncie (4)



Heurystyka mocno „oszukana” przebiegiem labiryntu.

Koniec nagrania 1

# Właściwości $A^*$

## Twierdzenie 1

$A^*$  jest zupełny (warunki jak dla UCS).

## Twierdzenie 2

Jeżeli  $h$  jest spójna, to  $A^*$  zwraca optymalną ścieżkę (wersja grafowa)

## Twierdzenie 3

Jeżeli  $h$  jest optymistyczna, to  $A^*$  w drzewie zwraca optymalną ścieżkę.

Dowody: będą, ale najpierw jeszcze trochę praktyki.

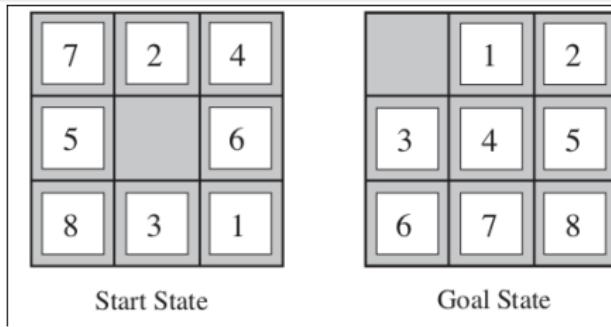
## Drobna uwaga

A\* oczywiście nie daje gwarancji znalezienia rozwiązania dla grafów nieskończonych, w których istnieją nieskończone ścieżki o skończonej sumie wag krawędzi  
Argument taki sam jak dla UCS

# Heurystyki dla ósemki

## Uwaga

Pewne aspekty tworzenia heurystyk można dość dobrze prześledzić na przykładzie ósemki



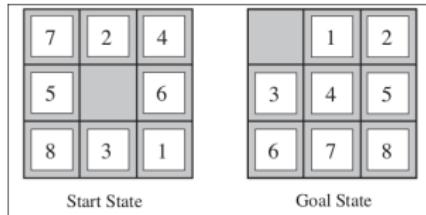
## Pytanie

Jak (optymistycznie) oszacować odległość tych dwóch stanów?

- Heurystyka musi być prosta do policzenia.
- Przy projektowaniu heurystyki kluczowe jest pilnowanie **optymizmu** (czyli że niedoszacujemy odległości).
- Choć teoretycznie wymagany jest silniejszy warunek (spójności), ale w praktyce naturalne optymistyczne heurystyki są spójne...
- ... a o optymizm łatwiej zadbać (i łatwiej przypomnieć sobie definicję)

**Zanim przewiniesz slajd spróbuj chwilę pomyśleć o optymistycznym szacowaniu liczby ruchów w ósemce**

# Heurystyki dla ósemki (2)



## Pomysł 1

Jak coś jest nie na swoim miejscu, to musi się ruszyć o co najmniej 1 krok. Zliczajmy zatem, ile kafelków jest poza punktem docelowym ( $h_1(s) = 8$ )

## Pomysł 2

Jak coś jest nie na swoim miejscu, to musi pokonać cały dystans do punktu docelowego. Zliczajmy zatem, ile kroków od celu jest każdy z kafelków ( $h_2(s) = 3 + 1 + 2 + 2 + 2 + 3 + 3 + 2 = 18$ )

## Pytanie

Która intuicyjnie jest lepsza?



- Intuicja mówi, że jeżeli coś **dokładniej** szacujemy, to algorytm bazujący na tych dokładniejszych szacunkach będzie działał lepiej
- Z dwóch optymistycznych heurystyk ta, która daje większe wartości, jest dokładniejsza.

- Dla  $h_2$  efektywność  $A^*$  jest 50000 razy większa niż IDS.
- Istnieją heurystyki dające jeszcze 10000 krotne przyspieszenie dla 15-ki, a milionowe dla 24-ki (wobec  $h_2$ )

Kiedy możliwy jest ruch w łamigłówce ósemka? Docelowe pole jest: (koniunkcja warunków):

- a) sąsiadujące
- b) wolne

Możemy rezygnować z części (lub wszystkich) warunków, otrzymując **łatwiejsze** łamigłówki.

## Uwaga

Liczba ruchów w łatwiejszym zadaniu od startu do punktu docelowego jest często sensowną heurystyką w zadaniu oryginalnym.

## Heurystyka $h_1$

Ruch możliwy jest **zawsze**.

## Heurystyka $h_2$

Ruch możliwy jest **gdy pole jest obok** (niekoniecznie puste).

## Heurystyka $h_3$

Ruch możliwy jest **gdy pole jest puste** (niekoniecznie obok).

# Relaksacja na mapie

Relaksacja w zadaniu poszukiwania w labiryntach lub przy podróży samochodem drogami:

Relaksacja w zadaniu poszukiwania w labiryntach lub przy podróży samochodem drogami:

- 1 W labiryncie: pomijanie ścian, czyli odległość taksówkową

Relaksacja w zadaniu poszukiwania w labiryntach lub przy podróży samochodem drogami:

- ① W labiryncie: pomijanie ścian, czyli odległość taksówkową
- ② W atlasie drogowym: pomijanie dróg, czyli odległość euklidesową (helikopterem)

# Operacja maksimum dla heurystyk

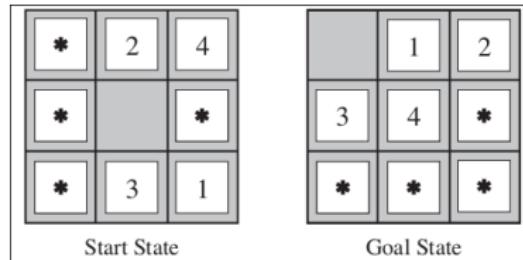
- Jak mamy dwie heurystyki  $h_1$  i  $h_2$ , obie optymistyczne
- to możemy zdefiniować  $h_3(x) = \max(h_1(x), h_2(x))$

## Uwaga

$h_3$  też będzie optymistyczna!

Heurystyki możemy budować korzystając z **baz wzorców**, zapamiętujących koszty rozwiązań **podproblemów** danego zadania.

Przykład:



- Najdujemy wszystkie podproblemy dla danego stanu (które mamy w bazie)
- A następnie bierzemy maksimum kosztów jako wartość heurystyki
- Możemy do tego maksimum dołożyć jakieś proste heurystyki (typu  $h_2$ ).

## Pytanie

A czy nie moglibyśmy użyć sumowania, zamiast maksimum?

## Działanie bazy wzorców (2)

- Niestety suma daje **niedopuszczalne** heurystyki (bo pewne ruchy liczymy wielokrotnie, gwiazdki w jednym wzorcu są istotnymi kafelkami w innym)
- **Pytanie:** Jak temu zapobiec?
- **Odpowiedź:** stosując „rozłączne” wzorce (nic się nie powtarza) i w każdym wzorcu liczyć tylko ruchy kafelków z liczbami.

To to są te najefektywniejsze heurystyki dla 8-ki

### Uwaga

Niemniej warto wiedzieć, że czasem rezygnuje się z optymalności i stosuje niedopuszczalne heurystyki (które czasem przeszacowują odległość), ze względu na szybkość działania.

Koniec nagrania II (lub III)

## Definicje

- $g(n)$  – koszt dotarcia do węzła  $n$
- $h(n)$  – szacowany koszt dotarcia od  $n$  do (najbliższego) punktu docelowego ( $h(s) \geq 0$ )
- $f(n) = g(n) + h(n)$

## Algorytm

Przeprowadź przeszukanie, wykorzystując  $f(n)$  jako priorytet węzła (czyli rozwijamy węzły od tego, który ma najmniejszy  $f$ ).

## Plan

Spróbujemy dowieść następujących rzeczy:

- ①  $A^*$  zwraca najkrótszą drogę
- ②  $A^*$  jest zupełny.

# Dowód optymalności

Potrzebujemy dwóch faktów:

- F1. Jeżeli  $h$  jest spójna, wówczas na każdej ścieżce wartości  $f$  są niemalejące.

D-d ( $n'$  jest następcą  $n$ ):

$$f(n') = g(n') + h(n') = g(n) + c(n, n') + h(n') \geq g(n) + h(n) = f(n)$$

- F2. Zawsze, gdy algorytm bierze węzeł do rozwinięcia, to koszt dotarcia do tego węzła jest optymalny (najmniejszy możliwy).

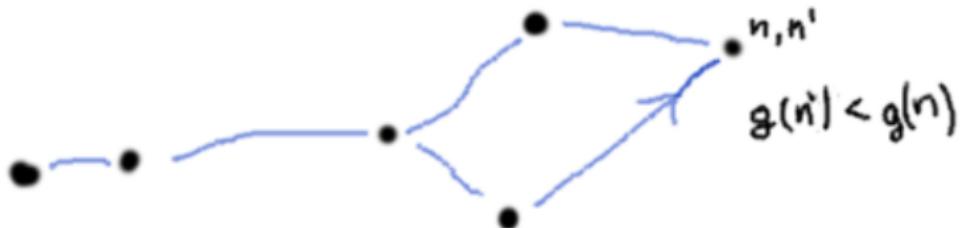
## Dowód F2



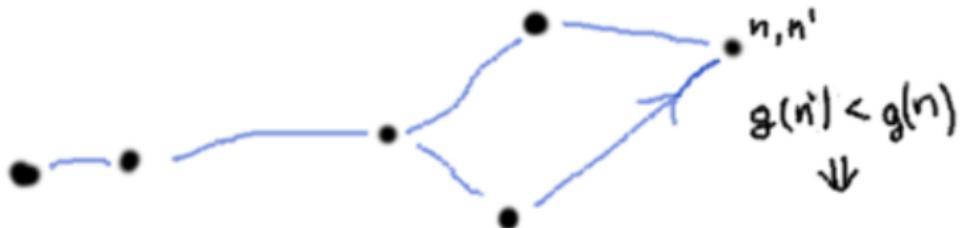
## Dowód F2



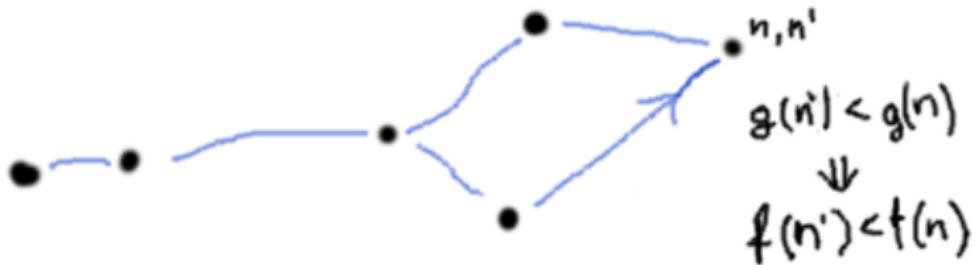
## Dowód F2



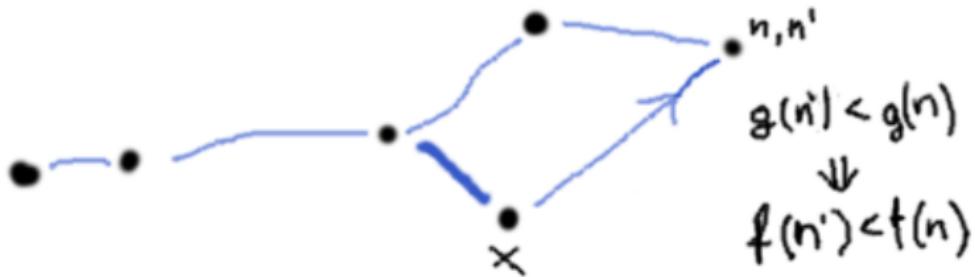
## Dowód F2



## Dowód F2

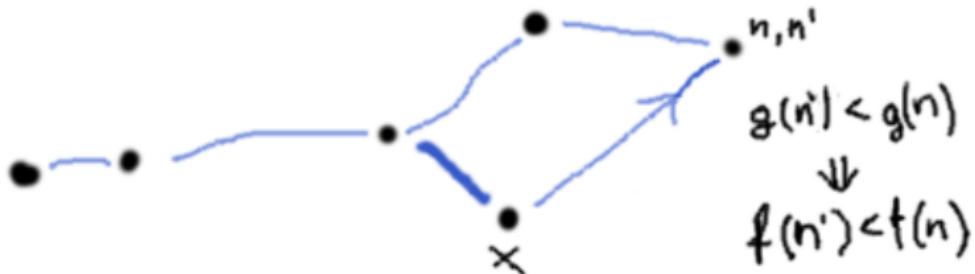


## Dowód F2



$$\leq F1: f(x) \leq f(n) < f(n')$$

## Dowód F2



$\leq F1: f(x) \leq f(n') < f(n)$

**SPRZECZNOŚĆ**

## Dowód F2

- Bierzemy nie wprost węzeł  $n$ , do którego kolejne dotarcie daje mniejszy koszt niż dotarcie pierwsze.
- Wartość  $f$  dla tego węzła drugi raz jest mniejsza ( $h$  takie same,  $g$  mniejsze)
- Na ścieżce od początku do  $n'$  drugiego mamy widziany w pierwszym przebiegu węzeł  $x$  (tuż po rozgałęzieniu)
- Z własności F1 mamy:  $f(x) < f(n)$

Zatem algorytm powinien wybrać  $x$  a nie  $n$ . **Sprzeczność.**

Popatrzmy na pierwszy znaleziony węzeł docelowy ( $n_{\text{end}}$ )

- $f(n_{\text{end}}) = g(n_{\text{end}}) + h(n_{\text{end}}) = g(n_{\text{end}})$  (bo  $h$  jest rozsądną)
- Każdy kolejny węzeł docelowy jest nielepszy, bo dla niego  
 $g(n) \geq g(n_{\text{end}})$

Niech  $C^*$  będzie kosztem najtańszego rozwiązania ( $g(n_{\text{end}})$ )

- Algorytm ogląda wszystkie węzły, dla których  $f(n) < C^*$
- Być może oglądnie również pewne węzły z konturu  $f(n) = C^*$ , przed wybraniem docelowego  $n$ , t.ż.  $f(n) = g(n) + 0 = C^*$

## Uwaga

Skończona liczba węzłów o  $g(n) \leq C^*$  gwarantuje to, że algorytm się skończy. Do skończości z kolei wystarczy założyć, że istnieje  $\varepsilon > 0$ , t.ż. wszystkie koszty są od niego większe bądź równe.

## Uwaga

$A^*$  nie rozwija węzłów t.ż.  $f(n) > C^*$ . Zatem im większa  $h$  (przy założeniu spełniania warunków dobrej heurystyki), tym lepsza.

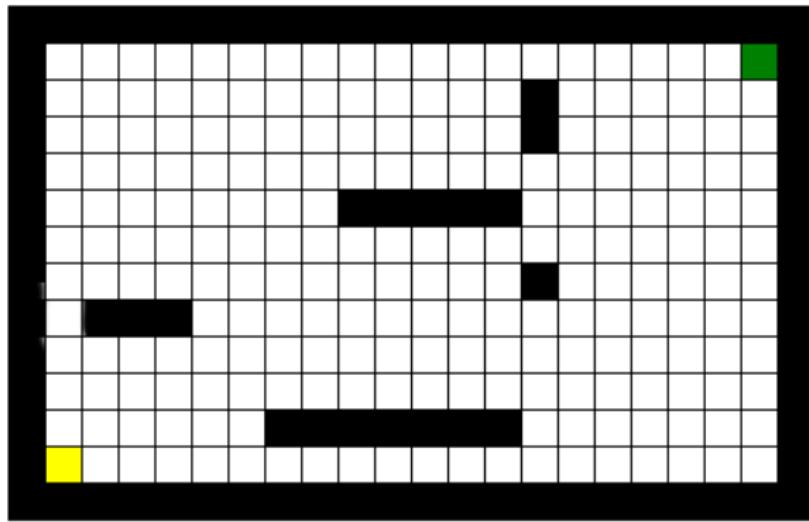
## Konsekwencja

Mając dwie spójne (optimistyczne) heurystyki  $h_1$  i  $h_2$ , możemy stworzyć  $h_3(n) = \max(h_1(n), h_2(n))$ , która będzie lepsza od swoich składników (szczegóły na ćwiczeniach).

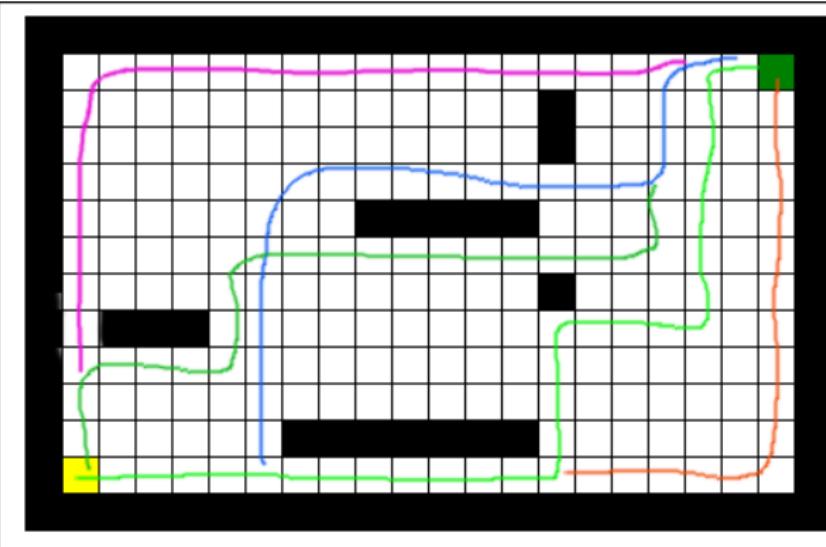
# Pytanie

Co się będzie działo, jeżeli nasza funkcja  $h$  będzie liczyła dokładną odległość od celu?

Bierzemy heurystykę Manhattańską (przymijmy, że cel jest jeden), czyli  $h(n) = |g_x - n_x| + |g_y - n_y|$



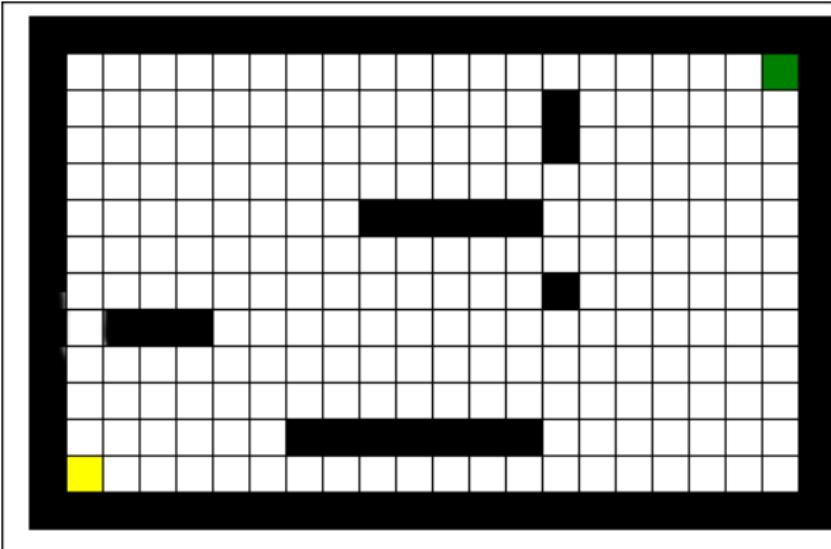
# Płaska funkcja $f$



Wszystkie ścieżki idące w prawo i do góry są optymalne. Funkcja  $f$  jest stała.

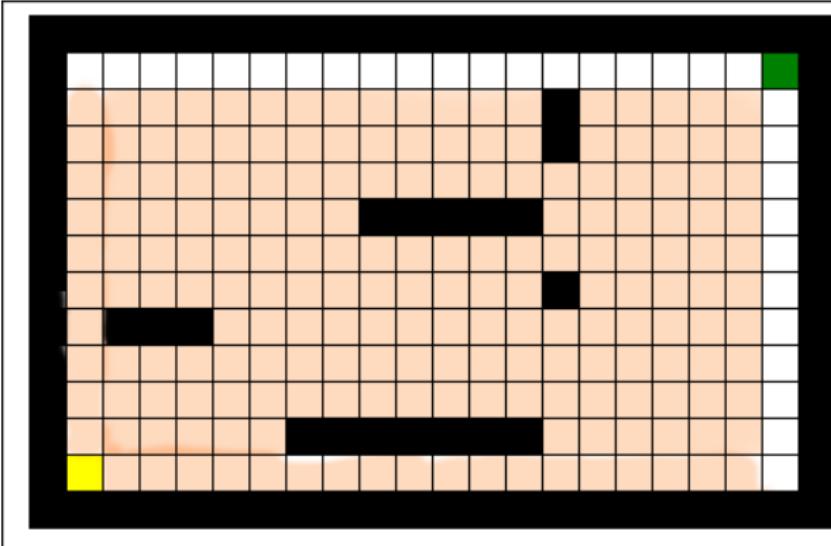
# Porównanie heurystyk

Porównajmy na przykładzie heurystykę manhattańską i euklidesową.  
Która jest lepsza? **Poniżej zaznaczone węzły, które na pewno  
musi obejrzeć  $A^*$  z heurystyką Euklidesową**

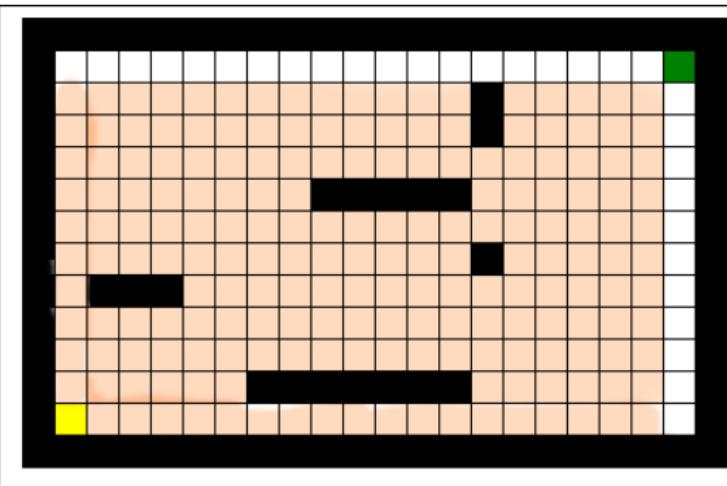


# Porównanie heurystyk

Porównajmy na przykładzie heurystykę manhattańską i euklidesową.  
Która jest lepsza? **Poniżej zaznaczone węzły, które na pewno  
musi obejrzeć  $A^*$  z heurystyką Euklidesową**



# Uzasadnienie



- Funkcja  $f$  z heurystyką  $h_M$  ma wszędzie wartość 30
- Funkcja  $f$  z heurystyką  $h_E$  osiąga wartość 30 jedynie na brzegach.

# Problemy więzowe

Paweł Rychlikowski

Instytut Informatyki UWr

18 marca 2021

ale najpierw jeszcze trochę o A\*

- Heurystyki mogą być niedopuszczalne (w szczególności, jeżeli są wynikiem uczenia się heurystyk)
- Oczywiście tracimy wówczas (w teorii i praktyce) gwarancje optymalności.
- Ale można otrzymać istotnie szybsze wyszukiwanie (o czym, mam nadzieję, przekonamy się na pracowni 2)

- **Pytanie:** Jaka jest najprostsza heurystyka niedopuszczalna (nieoptimistyczna)?
- **Odpowiedź:**  $(1 + \varepsilon)h(n)$ , gdzie  $h$  jest dopuszczalna

Czy ma ona jakiś sens?

# Heurystyki niedopuszczalne w praktyce

- **Pytanie:** Jaka jest najprostsza heurystyka niedopuszczalna (nieoptimistyczna)?
- **Odpowiedź:**  $(1 + \varepsilon)h(n)$ , gdzie  $h$  jest dopuszczalna

Czy ma ona jakiś sens?

Dla małego  $\epsilon$  będziemy rozstrzygać remisy oryginalnej funkcji  $f$  preferując węzły, które wydają się być bliższe celowi.

## Algorytm

Przeprowadź przeszukanie, wykorzystując  $f(n)$  jako priorytet węzła (czyli rozwijamy węzły od tego, który ma najmniejszy  $f$ ).

## Kluczowa właściwość

- ①  $A^*$  rozwija wszystkie węzły t.ż.  $f(n) < C^*$ .
- ②  $A^*$  rozwija niektóre węzły t.ż.  $f(n) = C^*$
- ③  $A^*$  nie rozwija węzłów t.ż.  $f(n) > C^*$ .

# Problemy więzowe

## Uwaga

Między ósemką a hetmanami jest istotna różnica (mimo, że oba można przedstawiać jako problemy przeszukiwania).

- W ósemce interesuje nam droga dotarcia do celu, który jest dobrze znany (i tym samym mało ciekawy)
- W hetmanach interesuje nas, jak wygląda cel – droga do niego może być dość trywialna (dostawianie po kolej poprawnych hetmanów, przestawianie hetmanów z losowego ustawnienia).

# Problem spełnialności więzów (2)

Problemy takie jak hetmany są:

- a) Bardzo istotne (ze względu na ich występowanie w rzeczywistym świecie)
- b) Na tyle specyficzne, że warto dla nich rozważać specjalne metody.

# Problemy spełnialności więzów. Definicja

## Definicja

Problem spełnialności więzów ma 3 komponenty:

1. Zbiór zmiennych  $X_1, \dots, X_n$
2. Zbiór dziedzin (przypisanych zmiennym)
3. Zbiór więzów, opisujących dozwolone kombinacje wartości jakie mogą przyjmować zmienne.

## Przykład

Zmienne:  $X, Y, Z$

Dziedziny:  $X \in \{1, 2, 3, 4\}$ ,  $Y \in \{1, 2\}$ ,  $Z \in \{4, 5, 6, 7\}$

Więzy:  $X + Y \geq Z$ ,  $X \neq Y$

# Komentarz do definicji

- Powyższy przykład to były więzy na dziedzinach skończonych, jeden z najważniejszych przypadków więzów.
- Ale można rozważyć inne dziedziny:
  - a) liczby naturalne, (trocę boli, że to nierostrzygalny problem)
  - b) liczby wymierne,
  - c) ciągi elementów, napisy
  - d) krotki
- Więzy określają relacje, często da się je wyrazić wzorem, ale nie jest to wymagane.

# Kolorowanie Australii



- Mamy pokolorować mapę Australii, za pomocą 3 kolorów: (**R, G, B**)
- Sąsiadujące prowincje muszą mieć różne kolory.

## Kolorowanie jako problem więzowy

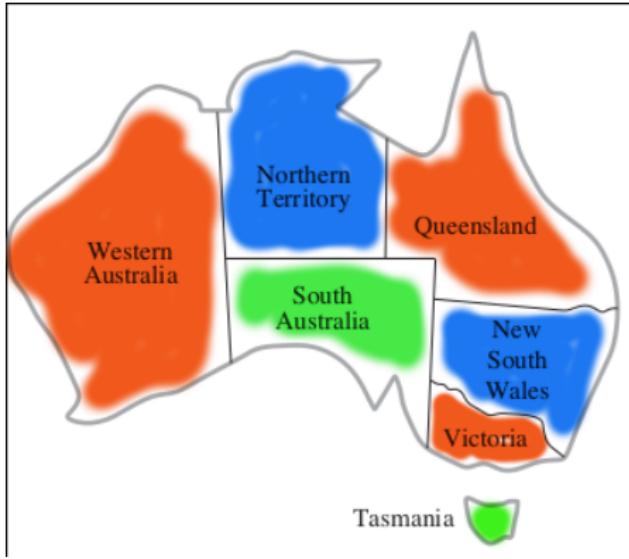
Zmienne:  $WA, NT, Q, NSW, V, SA, T$

Dziedziny:  $\{R, G, B\}$

Więzy:  $SA \neq WA, SA \neq NT, SA \neq Q, SA \neq NSW, SA \neq V, WA \neq NT, NT \neq Q, Q \neq NSW, NSW \neq V$



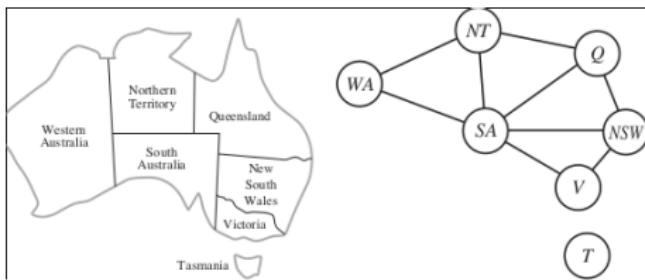
# Przykładowe kolorowanie



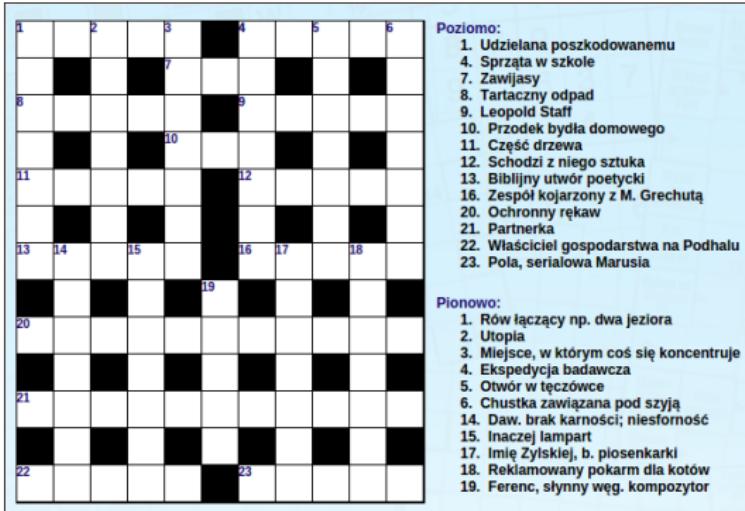
- Więzy (jako relacje) mogą mieć różną arność.
- **Unarne** – można potraktować jako modyfikację dziedziny (Tasmania nie jest czerwona) i zapomnieć.
- **Binarne** – jak w naszym przykładzie z kolorowaniem
- Mogą mieć też inną arność, w zasadzie dowolną (w praktyce spotyka się więzy o arności np. kilkaset)

# Graf więzów

- Dla więzów **binarnych** możemy stworzyć graf, w którym krawędź oznacza, że dwie zmienne są powiązane więzem.
- Więzy binarne są istotną klasą więzów, wiele algorytmów działa przy założeniu binarności więzów.



# Problemy dualne



Poziomo:

1. Udzielana poszkodowanemu
4. Sprząta w szkole
7. Zawijasy
8. Tartaczny odpad
9. Leopold Staff
10. Przodek bydła domowego
11. Część drzewa
12. Schodzi z niego sztuka
13. Biblijny utwór poetycki
16. Zespół kojarzony z M. Grechutą
20. Ochronny rękaw
21. Partnerka
22. Właściciel gospodarstwa na Podhalu
23. Pola, serialowa Marusia

Pionowo:

1. Rów łączący np. dwa jeziora
2. Utopia
3. Miejsce, w którym coś się koncentruje
4. Ekspedycja badawcza
5. Otwór w łyżeczce
6. Chustka zawiązana pod szyję
14. Daw. brak karności; niesformość
15. Inaczej lampart
17. Imię Zyłskiej, b. piosenkarki
18. Reklamowany pokarm dla kotów
19. Ferenc, słynny węg. kompozytor

## Pytanie

Co powinno być zmienną w zadaniu rozwiązywania krzyżówki?

# Krzyżówka (podstawowa)

Pomijamy (chwilowo?) kwestie zgodności hasła z definicją.



- Zmienne odpowiadają kratkom, dziedziną są znaki
- Więzy (fragment):  
jest-słowem-7(A,B,C,D,E,F,G), jest-słowem-3(B,H,I), ...

# Krzyżówka (dualna)

- Zmienne to słowa (dziedziną jest słownik przycięty do określonej długości)
- Mamy więc dla każdej pary krzyżujących się słów. Jaki?

Przykładowy więz  $C(w_1, w_2)$ :

- $w_1$  ma długość 6
- $w_2$  ma długość 10
- trzeci znak  $w_1$  jest taki sam, jak piąty znak  $w_2$

## Problemy dualne (2)

- Zwróćmy uwagę, że to, co zrobiliśmy z krzyżówką stosuje się do dowolnych więzów.
- Więzy w problemie prymarnym zmieniają się na zmienne w problemie dualnym (z dziedziną będącą dozwolonym zbiorem krotek)
- Dodatkowo potrzebujemy więzów, które mówią, że  $i$ -ty element jednej krotki jest  $j$ -tym elementem drugiej (te więzy są binarne!)

Koniec nagrania I

## Uwaga 1

To co odróżnia CSP od zadania przeszukiwania jest możliwość wykorzystania dodatkowej wiedzy o charakterze problemu do przeprowadzenia [wnioskowania](#).

## Uwaga 2

Podstawowym celem wnioskowania jest [zmniejszenie rozmiaru dziedzin](#) (a tym samym zmniejszenie przestrzeni przeszukiwań).

# Wnioskowanie. Przykład

## Przykład

Zmienne:  $X, Y, Z$

Dziedziny:  $X \in \{1, 2, 3, 4\}$ ,  $Y \in \{1, 2\}$ ,  $Z \in \{5, 6, 7, 8\}$

Więzy:  $X + Y \geq Z$ ,  $X \neq Y$

Czy możemy nie tracąc żadnego rozwiązania skreślić jakieś wartości z dziedzin?

Mogliśmy wywnioskować, że:  $X \in \{3, 4\}$ ,  $Y \in \{1, 2\}$ ,  $Z \in \{5, 6\}$

Jak widać, możemy też skreślić więz  $X \neq Y$

- Spójność (intuicyjnie) rozumiemy jako **niemożność wykreślenia żadnej wartości z dziedziny**.
- Mamy różne rodzaje spójności:
  - ① **Węzłowa** (każda wartość z dziedziny spełnia więzy unarne dla zmiennych)
  - ② **Łukowa**: jak dwie zmienne są połączone więzem, to dla każdej wartości z dziedziny X jest wartość w dziedzinie Y, t.ż. dla tych wartości więz jest spełniony.

## Uwaga

Są jeszcze inne rodzaje spójności. Więcej na ćwiczeniach.

# Spójność więzów. Przykład

**Więz:**  $X < Y$

**Dziedzina X:** {4, 6, 7, 10, 20}

**Dziedzina Y:** {1, 2, 4, 6, 7, 10}

**Brak spójności**

- Jeżeli weźmiemy X, to możemy wykreślić wartości 10, 20
- Jeżeli weźmiemy Y, to możemy wykreślić wartości 1,2, 4

Po wykreśleniu tych wartości warto przyjrzeć się innym więzom z X i Y.

## Definicja

Więz  $C$  dla zmiennych  $X$  i  $Y$  z dziedzinami  $D_X$  i  $D_Y$  jest **spójny łukowo**, wtt:

- Dla każdego  $x \in D_X$  istnieje takie  $y \in D_Y$ , że  $C(x, y)$  jest spełnione
- Dla każdego  $y \in D_Y$  istnieje takie  $x \in D_X$ , że  $C(x, y)$  jest spełnione

## Definicja

Więz  $C$  dla zmiennych  $X$  i  $Y$  z dziedzinami  $D_X$  i  $D_Y$  jest **spójny łukowo**, wtt:

- Dla każdego  $x \in D_X$  istnieje takie  $y \in D_Y$ , że  $C(x, y)$  jest spełnione
- Dla każdego  $y \in D_Y$  istnieje takie  $x \in D_X$ , że  $C(x, y)$  jest spełnione

**Sieć więzów** jest spójna łukowo, jeżeli każdy więz jest spójny łukowo.

Algorytm zapewnia spójność łukową sieci więzów.

## Idea

- ① Zarządzamy kolejką więzów,
- ② Usuwamy niepasujące wartości z dziedzin, analizując kolejne więzy z kolejki,
- ③ Po usunięciu wartości z dziedziny  $B$ , sprawdzamy wszystkie zmienne  $X$ , które występują w jednym więzie z  $B$

# Algorytm AC-3

---

**function**  $\text{AC-3}(csp)$  **returns** false if an inconsistency is found and true otherwise

**inputs:**  $csp$ , a binary CSP with components  $(X, D, C)$

**local variables:**  $queue$ , a queue of arcs, initially all the arcs in  $csp$

**while**  $queue$  is not empty **do**

$(X_i, X_j) \leftarrow \text{REMOVE-FIRST}(queue)$

**if**  $\text{REVISE}(csp, X_i, X_j)$  **then**

**if** size of  $D_i = 0$  **then return** false

**for each**  $X_k$  **in**  $X_i.\text{NEIGHBORS} - \{X_j\}$  **do**

add  $(X_k, X_i)$  to  $queue$

**return** true

---

**function**  $\text{REVISE}(csp, X_i, X_j)$  **returns** true iff we revise the domain of  $X_i$

$revised \leftarrow \text{false}$

**for each**  $x$  **in**  $D_i$  **do**

**if** no value  $y$  in  $D_j$  allows  $(x,y)$  to satisfy the constraint between  $X_i$  and  $X_j$  **then**

delete  $x$  from  $D_i$

$revised \leftarrow \text{true}$

**return**  $revised$

- Zwróćmy uwagę na niesymetryczność funkcji Revise (oznacza ona konieczność dodawania każdej pary zmiennych dwukrotnie)
- Zwróćmy uwagę, że istotny jest efekt uboczny tej funkcji: zmieniają się wartości dziedzin!

- Mamy  $n$  zmiennych, dziedziny mają wielkość  $O(d)$ . Mamy  $c$  więzów.
- Obsługa więzów to  $O(d^2)$
- Każdy więz może być włożony do kolejki co najwyżej  $O(d)$  razy.

## Złożoność

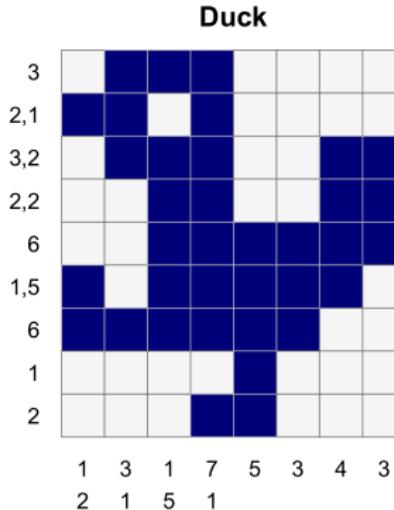
Złożoność wynosi zatem  $O(cd^3)$  (raczej pesymistycznie)

# Propagacja więzów

- Algorytm AC – 3 jest przykładowym algorytmem **propagacji więzów**
- Przeprowadzamy rozumowanie, które pozwala nam **bezpiecznie** usuwać zmienne z dziedziny.
- Zmniejszanie dziedziny zmiennej X może spowodować zmniejszenie dziedzin innych, związanych z nią zmiennych.

Koniec części II

# Spójność i obrazki logiczne



- Zmienne to wiersze i kolumny
- Spójność węzłowa (pojedynczy wiersz/kolumna zgodna ze specyfikacją)
- Spójność łukowa – zmiany w dziedzinie wierszu wpływają na kolumny i vice versa

Popatrzmy na przykład [inne okienko]

- (ubiegłoroczne) Zadanie z listy **Z1** pokazuje pewną technikę rozwiązywania zadań więzowych
- **Przypisuj wartości losowo i zarządzaj dziedzinami**

Wyjaśnienie w innym okienku

## Uwaga

Bardziej systematyczny sposób nawiązujący do tej metody nazywa się **backtrackingiem** (przeszukiwaniem z nawrotami)

przeszukiwanie z nawrotami = backtracking search

- Wariant przeszukiwania w głąb, w którym stanem jest **niepełne podstawienie**.
- Nie pamiętamy całej historii, ale potrafimy zrobić **undo**
- Po każdym przypisaniu wykonujemy jakąś formę **wnioskowania**, bo może da się zmniejszyć dziedziny...

# Backtracking

```
function BACKTRACK(assignment, csp) returns a solution, or failure
  if assignment is complete then return assignment
  var  $\leftarrow$  SELECT-UNASSIGNED-VARIABLE(csp)
  for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
    if value is consistent with assignment then
      add {var = value} to assignment
      inferences  $\leftarrow$  INFERENCE(csp, var, value)
      if inferences  $\neq$  failure then
        add inferences to assignment
        result  $\leftarrow$  BACKTRACK(assignment, csp)
        if result  $\neq$  failure then
          return result
      remove {var = value} and inferences from assignment
  return failure
```

# Backtracking. Uwagi

1. Możliwy jest też taki wariant, że najpierw uruchamiamy AC-3, potem Backtracking z jakimś uproszczonym wnioskowaniem.
2. Wnioskowanie może nie tylko wykreślać elementy z dziedziny, może również dodawać inne więzy (implikacje)

W wielu sytuacjach, jak mechanizm wnioskowania jest silny, to wykonywane jest bardzo niewiele **zgadnięć**.

- ① Jak wybieramy zmienną do podstawienia  
(`SelectUnassignedVariable`)
- ② W jakim porządku sprawdzamy dla niej wartości  
(`OrderDomainValue`)
- ③ Jak przeprowadzamy wnioskowanie (`Inference`)

# Przykład. Plan lekcji

- Rozmieszczamy lekcje: zajęcia otrzymują termin
- Mamy naturalne więzły:
  - Jeżeli  $Z_1$  i  $Z_2$  mają tego samego nauczyciela (klasę, salę), wówczas  $Z_1 \neq Z_2$
  - Nauczyciele nie mogą mieć zajęć o określonych porach (bo na przykład pracują w innych miejscach)
  - Wszystkie zajęcia klasy  $X$  danego dnia spełniają określone warunki: brak okienek, po jednej godzinie przedmiotu, itd.

## Pytanie

W jakiej kolejności rozmieszcza zajęcia Pani Sekretarka?

# Heurystyka: First Fail

## Definicja

Wybieramy tę zmienną, która **jest najtrudniejsza**, co oznacza, że:

- ma najmniejszą dziedzinę,
- występuje w największej liczbie więzów.

Inne nazwy: Most Constrained First, Minimum Remaining Values (MRV)

## Uzasadnienie

I tak będziemy musieli tę zmienną obsłużyć. Lepiej to zrobić, jak jeszcze inne zmienne są „wolne”

# Wybór wartości

- Wybieramy tę wartość, która w najmniejszym stopniu ogranicza przyszłe wybory **LCV**, Least Constraining Value.
- Przykład. W planie zajęć:
  1. Mamy teraz przydzielić termin zajęć panu A z klasą 1c
  2. Musimy później przydzielić zajęcia A z klasą 2a.
  3. Wcześniej przydzieliliśmy panią B z klasą 2a w czwartek na 8.
  4. **Jest to argument za tym, żeby (A,1c) też była na ósmą w czwartek** (bo nie stracimy żadnej możliwości dla (A,2a)).

# Wybór zmiennej vs wybór wartości

- W pierwszej chwili może dziwić przeciwnie traktowanie wyboru zmiennych i wartości.
- Celem FirstFail jest agresywne ograniczanie przestrzeni poszukiwań.
- Celem LCV jest dążenie do jak najszybszego znalezienia **pierwszego** rozwiązania.

Musimy rozpatrzyć wszystkie zmienne, ale niekoniecznie wszystkie wartości!

- Wybieramy **najgorszą** zmienną  
(ale każdą kiedyś musimy wybrać, a ta najgorsza najbardziej utrudni nam dalsze wybory)
- Wybieramy **najlepszą** wartość  
(ale często zależy nam na znalezieniu pierwszego rozwiązania, nie wszystkich)

## Uwaga

Możliwe inne **heurystyki** preferujące najbardziej obiecujące wartości! (można myśleć, że w tu jest miejsce na dowolny sensowny zachłanny algorytm wybierający wartość)

# O więzach ciąg dalszy

Paweł Rychlikowski

Instytut Informatyki UWr

24 marca 2021

## Definicja

Problem spełnialności więzów ma 3 komponenty:

1. Zbiór zmiennych  $X_1, \dots, X_n$
2. Zbiór dziedzin (przypisanych zmiennym)
3. Zbiór więzów, opisujących dozwolone kombinacje wartości jakie mogą przyjmować zmienne.

## Przykład

Zmienne:  $X, Y, Z$

Dziedziny:  $X \in \{1, 2, 3, 4\}$ ,  $Y \in \{1, 2\}$ ,  $Z \in \{4, 5, 6, 7\}$

Więzy:  $X + Y \geq Z$ ,  $X \neq Y$

- Czasami do problemu więzowego dodajemy dodatkowo zadanie maksymalizacji wartości pewnej funkcji:
  - Przydział robotników do maszyn spełniający określone wymagania i maksymalizujący produktywność.
  - Poprawny plan lekcji, maksymalizujący liczbę spełnionych miękkich wymagań nauczycieli (np. wolałbym nie mieć zajęć w piątek po 12, ale ...)

## Uwaga

W takich sytuacjach wybierając wartość bardzo często maksymalizujemy lokalne „zadowolenie” z rozwiązania.

# Super słaby backtracking

- Zwróćmy uwagę, że zachłanny algorytm wybierający zmienną (trudną) i wartość (obiecującą) jest np. algorytmem układania planu (nawet bez backtrackingu).
- Z drugiej strony przestrzeń jest tak ogromna (kilkaset zajęć, każde w kilkudziesięciu terminach), że trudno spodziewać się, aby backtracking dał sobie z nią radę (nawet backtracking na sterydach)

## Limited Discrepancy Search

**LDS** (**przeszukiwanie o ograniczonej rozbieżności**) jest wariantem przeszukiwania, w którym **jedynie  $d$  razy na całe przeszukanie**, mamy prawo wziąć nie pierwszy najlepszy termin, lecz drugi! ( $d$  jest małe, rzędu 1,2,3)

Można myśleć o tym tak:

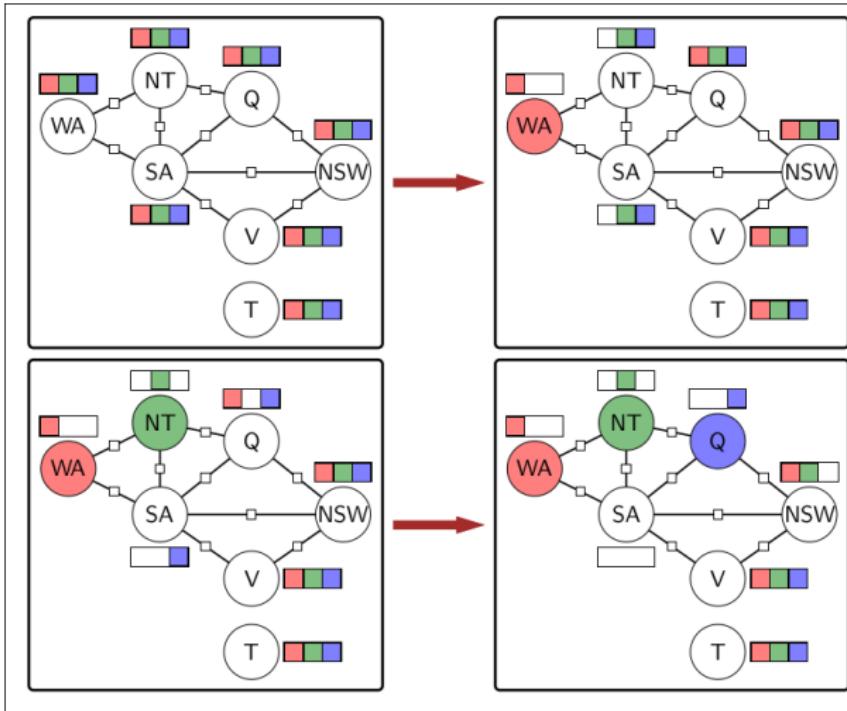
- Mamy ciąg decyzji (tyle ile zmiennych)
- Wybieramy  $d$  miejsc, w których podejmujemy „nieoptymalne” decyzje (z punktu widzenia heurystyki wyboru wartości)

- AC-3 może być kosztowne.
- Uproszczona forma: **Forward Checking**:
  - Zawsze, jak przypiszemy wartość, sprawdzamy, czy to przypisanie nie zmienia dziedzin innych zmiennych (które są w więzach z obsługiwana zmienną)
  - I tu zatrzymujemy wnioskowanie.

## Uwaga

Coś takiego można wykorzystać jako pełnoprawny algorytm.  
Wystarczy dodać jakąś losowość i restarty.

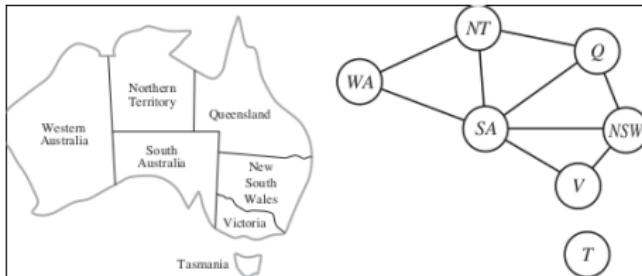
# Forward Checking – przykład



Źródło: CS221: Artificial Intelligence: Principles and Techniques

# First Fail w praktyce

- W kolorowaniu Australii wszystkie dziedziny na początku są równe...
- ale heurystyka First Fail w drugiej kolejności patrzy na liczbę więzów.



Wybór SA pozwala nam dalsze przeszukiwanie robić bez nawrotów.

# Więzy globalne (1)

- **Więzy globalne** to takie, które opisują relacje dużej liczby zmiennych (np. klasa nie ma okienek)
- Dobrym przykładem jest więz  $\text{alldifferent}(V_1, \dots, V_n)$

## Uwaga

Oczywiście da się wyrazić równoważny warunek za pomocą  $O(n^2)$  więzów  $V_i \neq V_j$ .

# Propagacja więzów globalnych

## Przykład

Mamy taką sytuację:  $X \in \{1, 2\}$ ,  $Y \in \{1, 2\}$ ,  $Z \in \{1, 2\}$ ,

Więzy:  $X \neq Y$ ,  $Y \neq Z$ ,  $X \neq Z$

- Spójny łukowo (niemożliwa propagacja)
- Globalne spojrzenie umożliwia stwierdzenie, że wartości **nie** **starczy**

Daje to prosty algorytm wykrywania sprzeczności więzów  
(porównanie sumy mnogościowej dziedzin i liczby zmiennych).

# Część II

- Przeszukiwanie lokalne nie próbuje systematycznie przeglądać przestrzeni rozwiązań (ogólniej: przestrzeni stanów)
- Zamiast tego pamięta jeden stan (lub niewielką, stałą liczbę stanów)
- Dla CSP stanem będzie kompletne przypisanie (niekoniecznie spełniające więzy).

# Problemy optymalizacyjne

- W tych problemach szukamy stanu, który maksymalizuje wartość pewnej funkcji (jakość planu).
- Często problemy z twardymi warunkami da się zamienić na problemy optymalizacyjne. Jak?

Można policzyć liczbę złych wierszy (kolumn) w obrazkach logicznych, albo liczbę szachów w hetmanach, albo....

## Uwaga

Możemy myśleć o spełnianiu CSP jako o zadaniu maksymalizacji liczby spełnionych więzów.

Możemy zatem stworzyć algorytm, w którym:

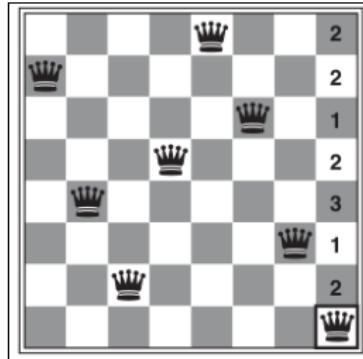
- Zmieniamy tę zmienną, która powoduje niespełnienie największej liczby więzów.
- Wybieramy dla niej wartość, która owocuje najmniejszą liczbą konfliktów.

# Przykład: 8 hetmanów

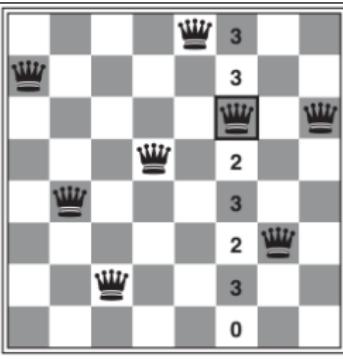
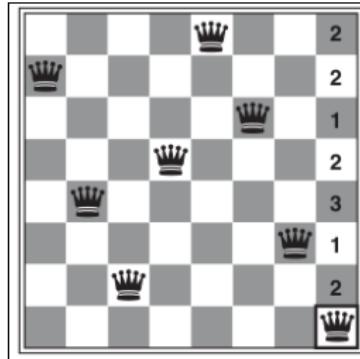
- Jak wybrać stan? (Wskazówka: powinniśmy umieć łatwo przejść ze stanu do stanu)
- Stan: w każdej kolumnie 1 hetman, Ruch: przesunięcie hetmana w górę lub w dół

Popatrzmy, jak działa [min-conflicts](#) dla hetmanów.

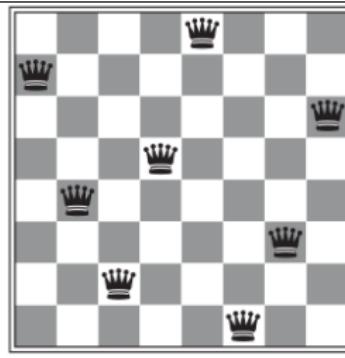
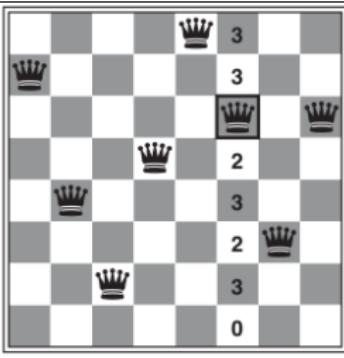
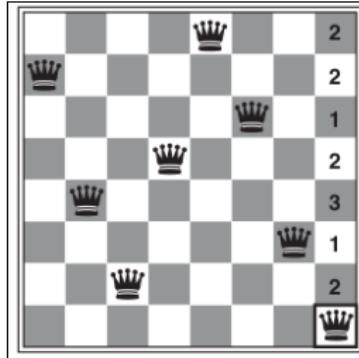
# Min-conflicts dla hetmanów



# Min-conflicts dla hetmanów



# Min-conflicts dla hetmanów



- Dla planszy  $8 \times 8$  osiąga sukces w 14% przypadków.
- Niby niezbyt dużo, ale możemy go uruchomić na przykład 20 razy, wówczas p-stwo sukcesu to ponad 95%.
- Można dopuszczać pewną liczbę ruchów w bok (czyli, że nie musimy poprawić, ale wystarczy, że nie pogorszymy, jak na obrazkach).
- Jak dopuścimy ruchy w bok , to wówczas mamy sukces w 94%

# Ważenie więzów

- Każdy więz ma wagę, początkowo wszystkie równe na przykład 1
- Waga więzów niespełnionych cały czas troszkę rośnie.
- Chcemy naprawiać nie zbiór więzów o liczności  $n$ , ale raczej zbiór więzów o największej sumarycznej wadze

Więzy trudne, rzadko spełniane będą miały coraz większy priorytet.

- Wyobraźmy sobie, że mamy problem, który się zmienia (ale w niewielkim stopniu)
- Przykład: obsługa linii lotniczych – bo zamykają się lotniska, pilot może złapać grypę, ...

## On-line CSP

Min-conflicts umożliwia rozwiązywanie tego typu zadań: stan początkowy to **ostatnie dobre** przypisanie.

# Część III

- Spróbujemy powiedzieć o programowaniu logicznym z więzami mówiąc maksymalnie mało o samym programowaniu logicznym
- o którym z kolei trochę powiemy, jak będziemy zajmowali się logiką.

## Uwaga

Możemy (na płytkim poziomie) potraktować CLP jako **constraint solver**, czyli system, w którym definiujemy zadanie więzowe i otrzymujemy rozwiązanie.

## Deklaratywne podejście do programowania

- Wypisujemy więzy (w jakimś formalnym języku)
- (możemy się wspomóc programowaniem, więzów może być dużo)
- Rozwiązaniem zajmuje się Solver (nie musimy implementować propagacji więzów i backtrackingu)

# Przykładowe systemy CLP

- SWI-Prolog (ma moduł clpfd)
- GNU-Prolog (trochę stary i nierozwijany)
- Eclipse (<http://eclipseclp.org/>)

- Zmienne FD (clpfd)
- Zmienne boolowskie (clpb)
- Zmienne rzeczywiste i wymierne (clpr)

Zajmiemy się tylko zmiennymi FD.

## clp(X)

Rozważa się również inne X-y: napisy, zbiory, przedziały.

Przypominamy: musimy określić zmienne, ich dziedziny oraz więzy na nich.

## Zmienne

Zmienne są zmiennymi prologowymi, piszemy je wielką literą.

## Dziedziny

`V in 1..10`

`[A,B,C,D] ins 1..10`

## Więzy

Języki CLP mają bardzo bogate możliwości wyrażania problemów za pomocą więzów.

# Przyślijcie Więcej Pieniądzy



```
puzzle(Vars) :-  
    Vars = [S,E,N,D,M,O,R,Y],  
    Vars ins 0..9,  
    all_different(Vars),  
    S*1000 + E*100 + N*10 + D +  
    M*1000 + O*100 + R*10 + E #=  
    M*10000 + O*1000 + N*100 + E*10 + Y,  
    M #\= 0, S #\= 0.
```

# Wykorzystanie solwera więzowego

## Postać programu CLP

```
name([V1, ..., Vn]) :-  
    V1 in 1..K, ..., Vn in 1..K,  
    V1 # >= V5, abs(V2+V6) # = abs(V7-V8)  
    min(V3,V7) # > V3 + 2*V1  
    labeling([options], [V1, ..., Vn]).
```

# Wykorzystanie solwera więzowego

## Postać programu CLP

```
name([V1, ..., Vn]) :-  
    V1 in 1..K, ..., Vn in 1..K,  
    V1 # >= V5, abs(V2+V6) # = abs(V7-V8)  
    min(V3,V7) # > V3 + 2*V1  
    labeling([options], [V1, ..., Vn]).
```

- Czyli mamy obsługę **zmiennych i dziedzin**, **ustanowienie więzów** oraz wywołanie przeszukiwania z nawrotami (**labeling**), a na końcu wywołanie głównego predykatu.

## Postać programu CLP

```
name([V1, ..., Vn]) :-  
    V1 in 1..K, ..., Vn in 1..K,  
    V1 # >= V5, abs(V2+V6) # = abs(V7-V8)  
    min(V3,V7) # > V3 + 2*V1  
    labeling([options], [V1, ..., Vn]).
```

- Czyli mamy obsługę **zmiennych i dziedzin**, **ustanowienie więzów** oraz wywołanie przeszukiwania z nawrotami (**labeling**), a na końcu wywołanie głównego predyktu.
- Część czarna jest częścią *techniczną*, stanowiącą naszą *daninę* dla Prologa

# Wykorzystanie solwera więzowego

## Postać programu CLP

```
name([V1, ..., Vn]) :-  
    V1 in 1..K, ..., Vn in 1..K,  
    V1 # >= V5, abs(V2+V6) # = abs(V7-V8)  
    min(V3,V7) # > V3 + 2*V1  
    labeling([options], [V1, ..., Vn]).
```

- Czyli mamy obsługę **zmiennych i dziedzin**, **ustanowienie więzów** oraz wywołanie przeszukiwania z nawrotami (**labeling**), a na końcu wywołanie głównego predyktu.
- Część czarna jest częścią *techniczną*, stanowiącą naszą *daninę* dla Prologa
- Ten program możemy napisać, używając **Ulubionego Języka Programowania** – wystarczy, że ma **print**, **printf**, **puts**, ...

## Przykład

Popatrzmy, jak to działa dla zadania z N hetmanami.

- Warunki określające dziedziny:

```
def domains(Qs, N):
    return [ q + ' in 0..' + str(N-1) for q in Qs ]
```

- Brak szachów w poziomie (alldifferent)

```
def all_different(Qs):
    return ['all_distinct([' + ', '.join(Qs) + '])']
```

- Brak szachów po przekątnej

```
def diagonal(Qs):
    N = len(Qs)
    return [ "abs(%s - %s) #\=\= abs(%d-%d)" % (Qs[i],Qs[j],i,j)
            for i in range(N) for j in range(N) if i != j ]
```

# Pajtono-prolog (2)

Sklejenie wszystkich części:

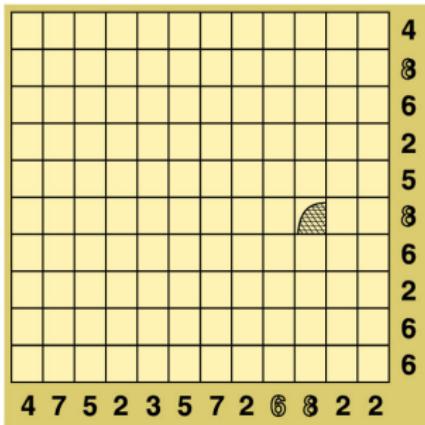
---

```
def queens(N):
    vs = ['Q' + str(i) for i in range(N)]
    print ':- use_module(library(clpf)).'
    print 'solve([' + ', '.join(vs) + ']) :- '
    cs = domains(vs, N) + all_different(vs) + diagonal(vs)
    print_constraints(cs, 4, 70),
    print
    print '    labeling([ff], [' + commas(vs) + ']).'
    print
    print ':- solve(X), write(X), nl.'
```

- Zobaczmy, jak działa program `queen_produce.py`
- Jak wyglądają wynikowe programy
- Jak duże instancje jesteśmy w stanie rozwiązywać?

## Przykład 2: burze

Może pojawią się na liście P3...

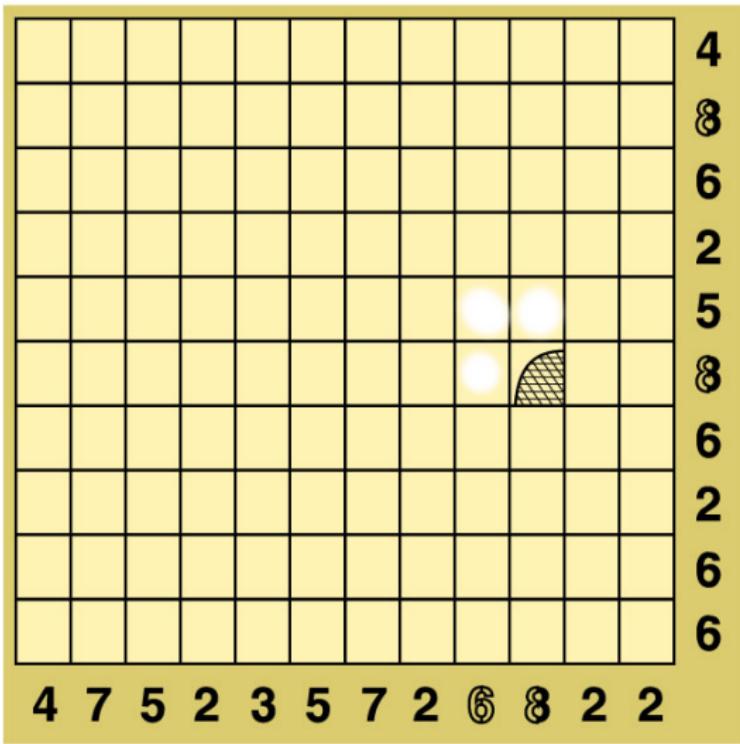


### Zasady

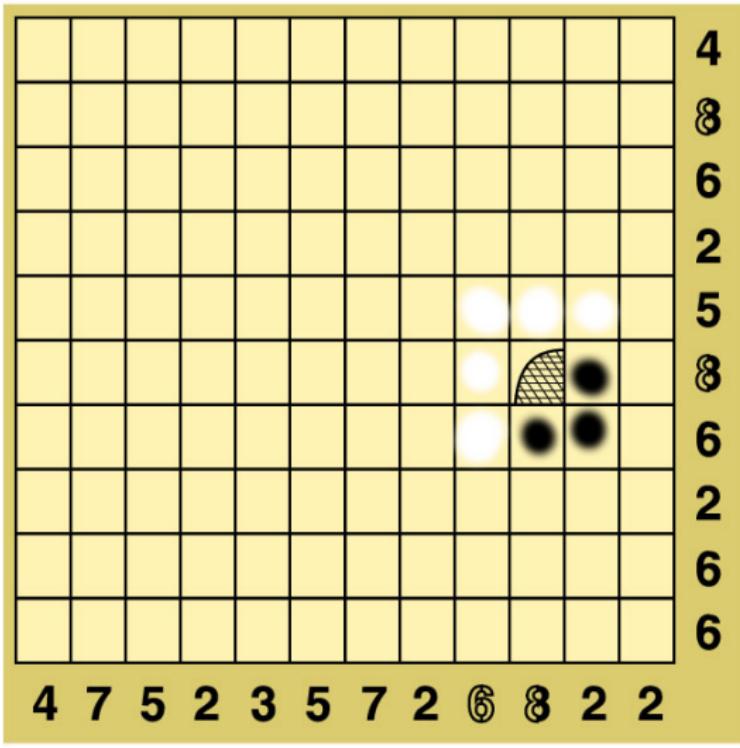
1. Radary mówią, ile jest pól burzowych w wierszach i kolumnach.
2. Burze są prostokątne.
3. Burze nie stykają się rogami.
4. Burze mają wymiar co najmniej  $2 \times 2$ .



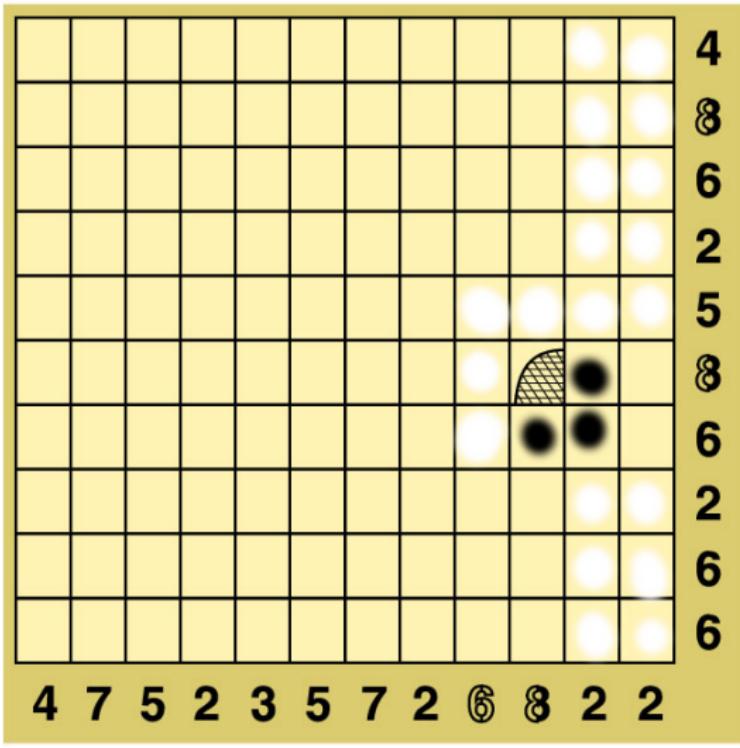
# Burze: wnioskowanie



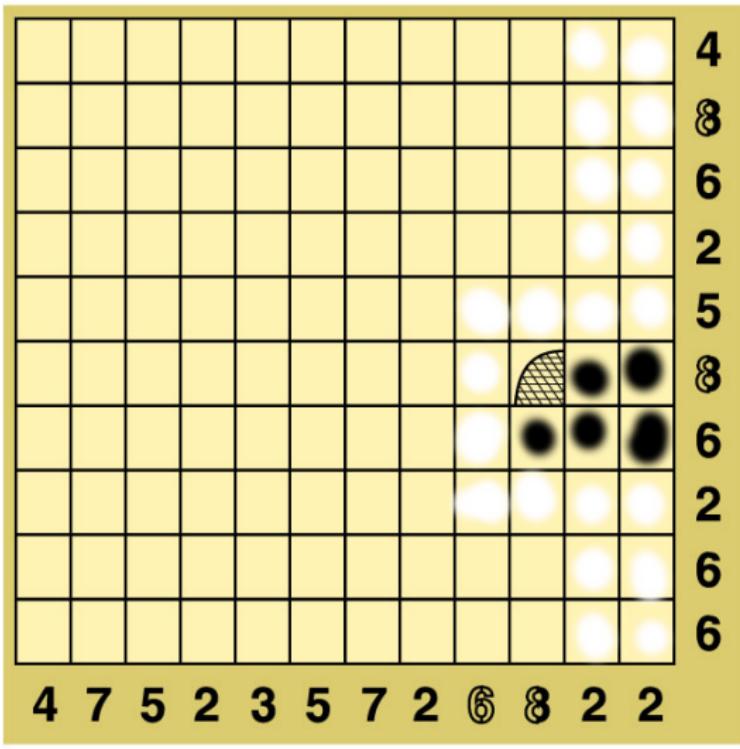
# Burze: wnioskowanie



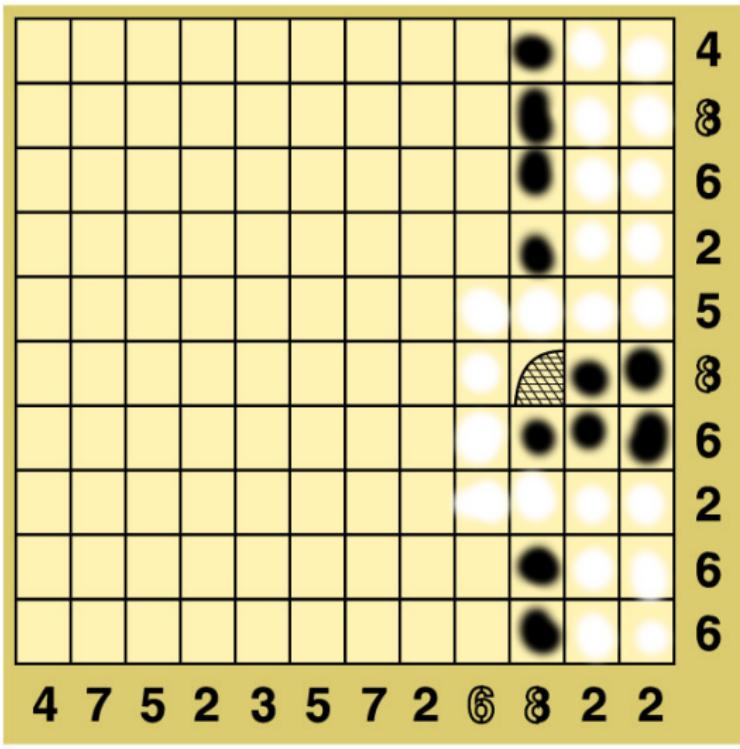
# Burze: wnioskowanie



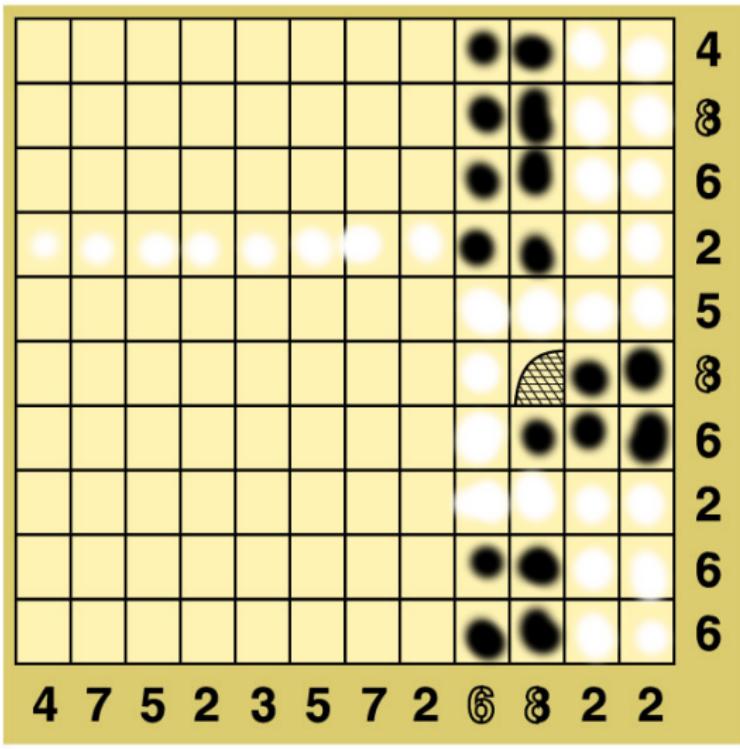
# Burze: wnioskowanie



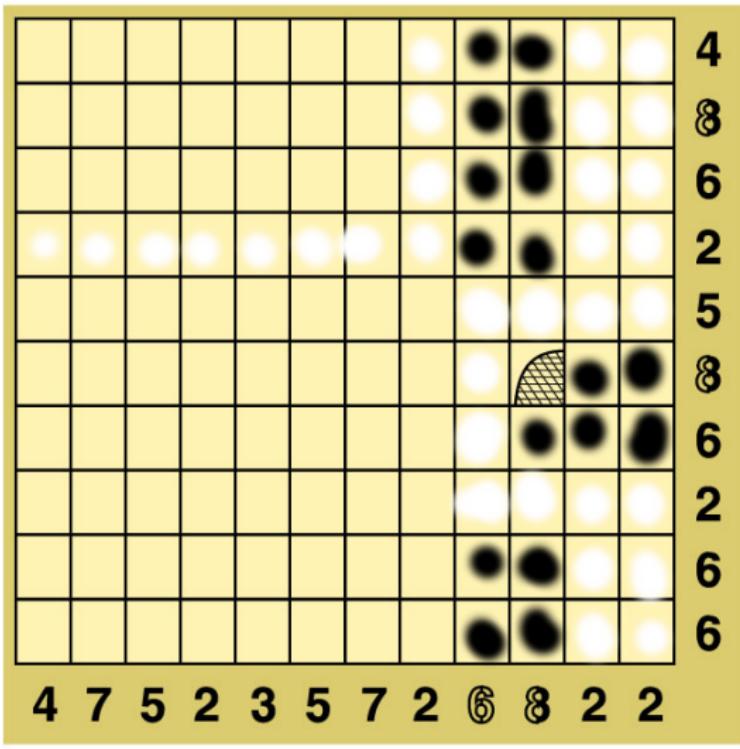
# Burze: wnioskowanie



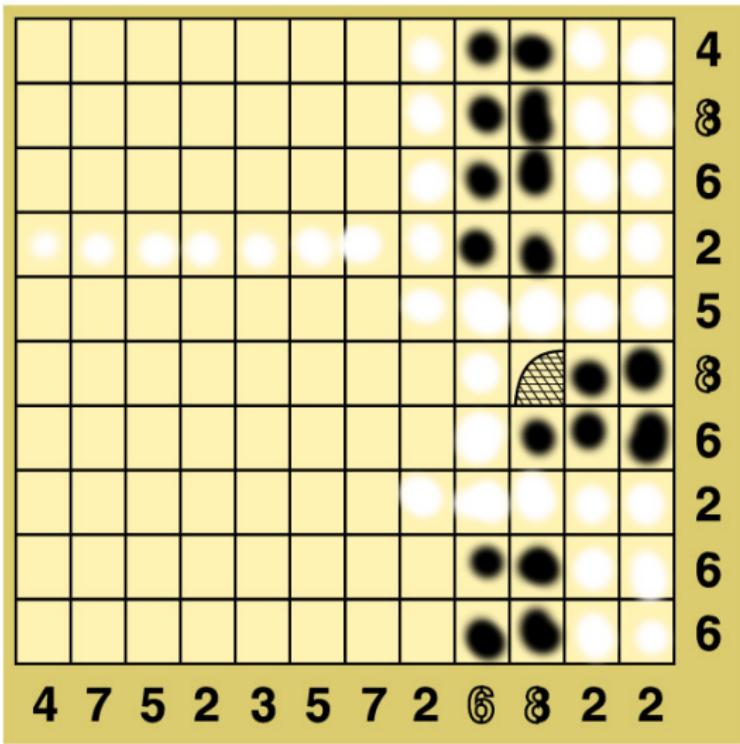
# Burze: wnioskowanie



# Burze: wnioskowanie



# Burze: wnioskowanie



# Rozwiązanie

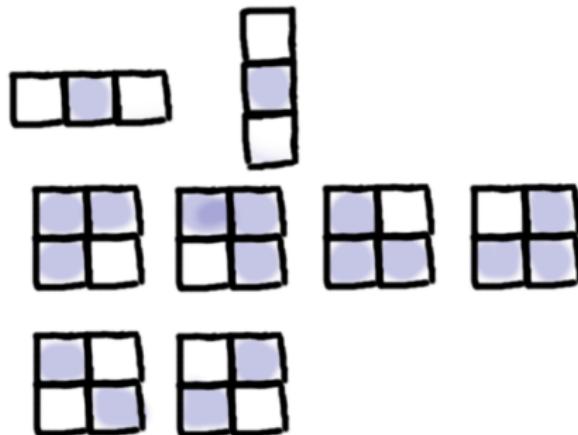
- Strategia 1: jak obrazki logiczne, + wnioskowanie
- Strategia 2: wykorzystujemy SWI-Prolog

# Kodowanie burz

- Zmienne, dziedziny: piksele, 0..1
- Radary:  $b_1 + b_2 + \cdots + b_n = K$
- Prostokąty: ?
- Co najmniej  $2 \times 2$ ?
- Nie stykają się rogami.

- Jak wygląda **każdy** kwadrat  $2 \times 2$ ?
- Jak wygląda **każdy** prostokąt  $1 \times 3$  albo  $3 \times 1$ ?

## Zabronione układy



## Pytanie

Jak wyrazić to językiem relacji arytmetycznych?

# Warunek dobrych 3 pól

Mamy zmienne  $A, B, C$

- $A + 2B + 3C \neq 2$
- $B \times (A + C) \neq 2$

## Naturalne sformułowanie

Jeżeli środkowy piksel jest ustawiony, to wówczas przynajmniej 1 z otaczających go jest jedynką.

$$B \Rightarrow (A + C > 0)$$

- Inny przykład:  $A \#<=> B \#> C$
- Naturalna propagacja:
  - Ustalenie  $A$  dorzuca więz
  - Jak wiemy, czy prawdziwy jest  $B \#> C$ , to znamy wartość  $A$

## tuples\_in

Wymieniamy explicite krotki wartości, jakie może przyjmować krotka zmiennych

## Uwaga

Zauważmy, że ten więz pasuje do lokalnych warunków dla burz, na przykład dla prostokątów  $3 \times 1$ :

```
tuple_in( [A,B,C], [ [0,0,0], [1,1,0], [1,0,0] ,  
[0,1,1], [0,0,1], [1,1,1], [1,0,1] ] )
```

# Przeszukiwanie lokalne i gry

Paweł Rychlikowski

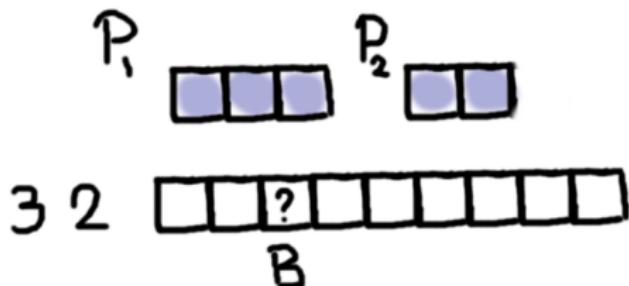
Instytut Informatyki UWr

23 kwietnia 2021

Najpierw jeszcze trochę o więzach

- Inny przykład:  $A \#<=> B \#> C$
- Naturalna propagacja:
  - Ustalenie  $A$  dorzuca więz
  - Jak wiemy, czy prawdziwy jest  $B \#> C$ , to znamy wartość  $A$

# Reifikacja i obrazki logiczne



- Użycie zmiennych  $P_1$  i  $P_2$  określających położenie bloku pozwala zmniejszyć dziedziny ( $|P_1| + |P_2|$  zamiast  $|P_1| \times |P_2|$ ) (mniejsze zużycie pamięci, niezmniejszona liczba kombinacji)
- Zmienna  $B$  ma wartość logiczną:  
*3 jest przykryte przez blok rozpoczęty w  $P_1$  lub przez blok rozpoczęty w  $P_2$*

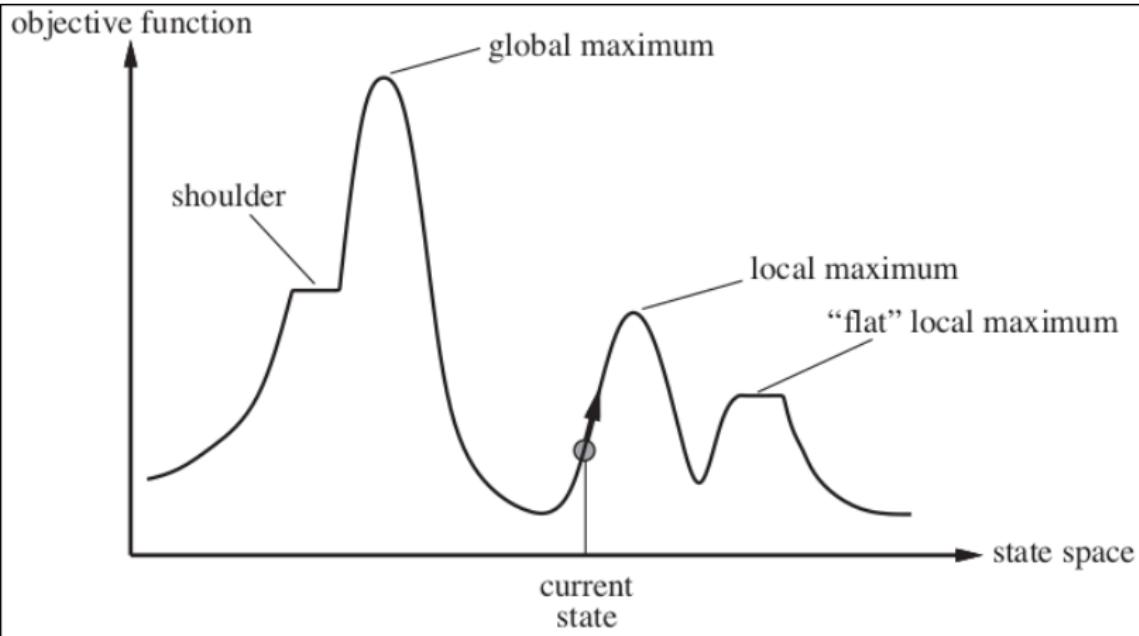
- Metoda z porzedniego slajdu jest bardzo ogólna!
- **Opis zadania prawie automatycznie przekłada się na algorytm!**

Na tym skończymy o więzach i  
przejdziemy do przeszukiwania  
lokalnego

# Przeszukiwania lokalne (ogólnie)

- Powiemy sobie o paru ideach związanych z przeszukiwaniem lokalnym.
- Można je wykorzystywać w zadaniach więzowych (MinConflicts z poprzedniego wykładu), ale nie tylko.

# Krajobraz przeszukiwania lokalnego



# Czym może być funkcja, którą minimalizujemy?

- 1 Liczbą **niespełnionych** więzów.

# Czym może być funkcja, którą minimalizujemy?

- ① Liczbą **niespełnionych** więzów.
- ② **Wagą** niespełnionych więzów.

# Czym może być funkcja, którą minimalizujemy?

- ① Liczbą **niespełnionych** więzów.
- ② **Wagą** niespełnionych więzów. Porównaj więzy:
  - ① Nauczyciel ma tylko z jedną klasą lekcje na raz
  - ② nikt nie ma dwóch biologii jednego dnia.

# Czym może być funkcja, którą minimalizujemy?

- ① Liczbą **niespełnionych** więzów.
- ② **Wagą** niespełnionych więzów. Porównaj więzy:
  - ① Nauczyciel ma tylko z jedną klasą lekcje na raz
  - ② nikt nie ma dwóch biologii jednego dnia.
- ③ Czymś niezwiązanym bezpośrednio z więzami

# Czym może być funkcja, którą minimalizujemy?

- ① Liczbą **niespełnionych** więzów.
- ② **Wagą** niespełnionych więzów. Porównaj więzy:
  - ① Nauczyciel ma tylko z jedną klasą lekcje na raz
  - ② nikt nie ma dwóch biologii jednego dnia.
- ③ Czymś niezwiązanym bezpośrednio z więzami
  - produktywnością zespołu robotników (maksymalizemy, nie minimalizujemy!)

# Czym może być funkcja, którą minimalizujemy?

- ① Liczbą **niespełnionych** więzów.
- ② **Wagą** niespełnionych więzów. Porównaj więzy:
  - ① Nauczyciel ma tylko z jedną klasą lekcje na raz
  - ② nikt nie ma dwóch biologii jednego dnia.
- ③ Czymś niezwiązanym bezpośrednio z więzami
  - produktywnością zespołu robotników (maksymalizemy, nie minimalizujemy!)
  - zadowoleniem gości weselnych z towarzystwa przy stolikach,

# Czym może być funkcja, którą minimalizujemy?

- ① Liczbą **niespełnionych** więzów.
- ② **Wagą** niespełnionych więzów. Porównaj więzy:
  - ① Nauczyciel ma tylko z jedną klasą lekcje na raz
  - ② nikt nie ma dwóch biologii jednego dnia.
- ③ Czymś niezwiązanym bezpośrednio z więzami
  - produktywnością zespołu robotników (maksymalizemy, nie minimalizujemy!)
  - zadowoleniem gości weselnych z towarzystwa przy stolikach,
  - potencjalnym zyskiem sklepu,

# Czym może być funkcja, którą minimalizujemy?

- ① Liczbą **niespełnionych** więzów.
- ② **Wagą** niespełnionych więzów. Porównaj więzy:
  - ① Nauczyciel ma tylko z jedną klasą lekcje na raz
  - ② nikt nie ma dwóch biologii jednego dnia.
- ③ Czymś niezwiązanym bezpośrednio z więzami
  - produktywnością zespołu robotników (maksymalizemy, nie minimalizujemy!)
  - zadowoleniem gości weselnych z towarzystwa przy stolikach,
  - potencjalnym zyskiem sklepu,
  - dopasowaniem do danych uczących

## Uwaga

Ważna część uczenia maszynowego dotyczy **maksymalizacji dopasowania do danych uczących**

Hill climbing jest chyba najbardziej naturalnym algorytmem inspirowanym poprzednim rysunkiem.

- Dla stanu znajdujemy wszystkie następcy i wybieramy ten, który ma największą wartość.
- Powtarzamy aż do momentu, w którym nie możemy nic poprawić

## Problem

Oczywiście możemy utknąć w lokalnym maksimum.

## Uwaga

Możemy podjąć dwa działania, oba testowaliśmy w obrazkach logicznych:

- ① Dorzucać ruchy niekoniecznie poprawiające (losowe, ruchy **w bok**)
- ② Gdy nie osiągamy rozwiązania przez dłuższy czas rozpoczętym od początku.

Hill climbing + random restarts (w trywialny sposób) jest algorytmem zupełnym z p-stwem 1 (bo „kiedyś” wylosujemy układ startowy)

- a) **Stochastic hill climbing** – wybieramy losowo ruchy w górę (p-stwo stałe, albo zależne od wielkości skoku).
- b) **First choice hill climbing** – losujemy następnika tak dugo, aż będzie on ruchem w góre
  - dobre, jeżeli następników jest bardzo dużo

## Uwaga

Idee z tego i kolejnych algorytmów można dowolnie mieszać – na pewno coś wyjdzie!

- Motywacja fizyczna: ustalanie struktury krystalicznej metalu.
- Jeżeli będziemy ochładzać powoli, to metal będzie silniejszy (blizej globalnego minimum energetycznego).
- **Symulowane wyżarzanie** – próba oddania tej idei w algorytmie.

## Algorytm

Symulujemy opadającą temperaturę, prawdopodobieństwo ruchu chaotycznego zależy **malejąco** od temperatury.

## Symulowane wyżarzanie (2)

- Przykładowa implementacja bazuje na [first choice hill climbing](#).
- Jak wylosowany ruch ( $\text{r}$ ) jest lepszy (czyli  $\Delta F > 0$ ), to go wykonujemy (maksymalizacja  $F$ ).
- W przeciwnym przypadku wykonujemy ruch  $\text{r}$  z p-stwem  
 $p = e^{\frac{\Delta F}{T}}$
- Pilnujemy, żeby  $T$  zmniejszało się w trakcie działania (i było cały czas dodatnie)

### Komentarze do wzoru

- $\Delta F \leq 0, T > 0$ , czyli  $0 \leq p \leq 1$ .
- Im większe pogorszenie, tym mniejsze p-stwo
- Im większa temperatura, tym większe p-stwo.

## Problem

Być może płaskie maksimum lokalne.

## Rozwiązanie

Dodajemy pamięć algorytmowi, zabraniamy powtarzania ostatnio odwiedzanych stanów.

- Zamiast pamiętać pojedynczy stan, pamiętamy ich  $k$  (wiązkę).
- Generujemy następcy dla każdego z  $k$  stanów.
- Pozostawiamy  $k$  liderów.

## Uwaga 1

To nie to samo co  $k$  równoległych wątków hill-climbing (bo uwaga algorytmu może przerzucać się do bardziej obiecujących kawałków przestrzeni)

## Uwaga 2

Beam search jest bardzo popularnym algorytmem w różnych zadaniach wykorzystujących sieci neuronowe do modelowania sekwencji (np. tłumaczenie maszynowe).

- Zarządzamy **populacją** osobników (czyli np. pseudorozwiązań jakiegoś problemu więzowego).
- Mamy dwa rodzaje operatorów:
  - a) Mutacja, która z jednego osobnika robi innego, podobnego.
  - b) Krzyżowanie, która z dwóch osobników robi jednego, w jakiś sposób podobnego do „rodziców”.
- Nowe osobniki oceniane są ze względu na wartość **funkcji przystosowania**
- Przeżywa *k* najlepszych.

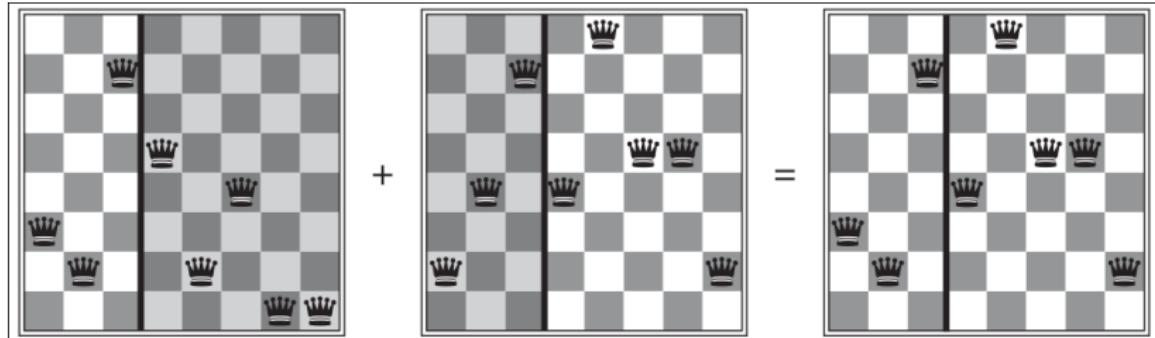
## Uwaga

Zauważmy, że choć zmienił się język, jeżeli pominiemy krzyżowanie, to otrzymamy wariant Local beam search (mutacja jako krok w przestrzeni stanów).

# Krzyżowanie. Przykład

## Pytanie

Czym mogłyby być krzyżowanie dla zadania z  $N$  hetmanami?



# Algorytmy ewolucyjne. Kilka uwag

- ① Krzyżowanie i mutacje można zorganizować tak, że najpierw powstają dzieci, a następnie się mutują z pewnym prawdopodobieństwem.
- ② Wybór osobników do rozmnażania może zależeć od funkcji dopasowania (większe szanse na reprodukcję mają lepsze osobniki)
- ③ Można mieć wiele operatorów krzyżowania i mutacji.

Koniec części I

Rozpoczynamy nowy wątek wykładu

# Przeszukiwanie w grach

# Przykładowa gra

- Gracz **A** wybiera jeden z trzech zbiorów:
  1.  $\{-50, 50\}$
  2.  $\{1, 3\}$
  3.  $\{-5, 15\}$
- Następnie gracz **B** wybiera liczbę z tego zbioru.

## Pytanie

Co powinien zrobić **A**, żeby uzyskać jak największą liczbę?

## Nasza gra

- 1.  $\{-50, 50\}$
- 2.  $\{1, 3\}$
- 3.  $\{-5, 15\}$

Racjonalny wybór dla **A** zależy od (modelu) gracza **B**

- Współpracujący: Oczywiście 1.
- Losowy ( $z p = \frac{1}{2}$ ) Wybór 3 (średnio 5)
- „Złośliwy” :wybór 2 (gwarantujemy wartość 1)

# Wyszukiwanie w grach

- Nieco inna rodzina zadań wyszukiwania, w których mamy dwóch (lub więcej) agentów.
- Interesy agentów są (przynajmniej częściowo) rozbieżne.
- Rozgrywka przebiega w turach, w których gracze na zmianę wybierają swoje ruchy.

## Definicja

Gra jest problemem przeszukiwania, zadanym przez następujące składowe:

- ① Zbiór stanów, a w nim  $S_0$ , czyli stan początkowy
- ②  $\text{player}(s)$ , funkcja określająca gracza, który gra w danym stanie.
- ③  $\text{actions}(s)$  – zbiór ruchów możliwych w stanie  $s$
- ④  $\text{result}(s,a)$  – funkcja zwracająca stan powstały w wyniku zastosowania akcji  $a$  w stanie  $s$ .
- ⑤  $\text{terminal}(s)$  – funkcja sprawdzająca, czy dany stan kończy grę.
- ⑥  $\text{utility}(s, \text{player})$  – funkcja o wartościach rzeczywistych, opisująca wynik gry z punktu widzenia danego gracza.

# Gra o sumie zerowej

## Definicja

W **grze o sumie zerowej** suma wartości stanów terminalnych dla wszystkich graczy jest stała (niekoniecznie zera, ale...)

## Konsekwencje:

- Zysk jednego gracza, jest stratą drugiego.
- Kooperacja nic nie daje.

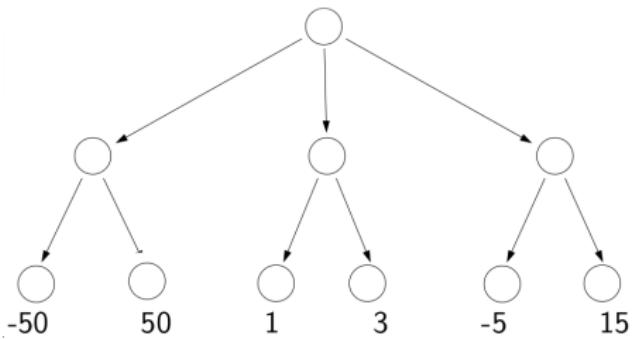
## Uwaga

Zaczniemy od gier o sumie zerowej i gracza, wcześniej nazwanego **złośliwym** (lepiej go nazwać **racjonalnym**)

# Różnice między grami a zwykłym przeszukiwaniem

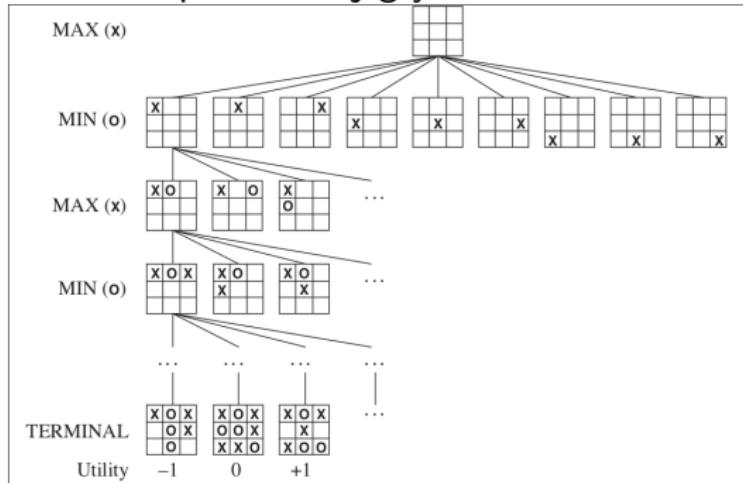
- ① Mamy graczy: stan gry wskazuje na gracza, który ma się ruszać.
- ② Stany końcowe mają wartości, różne dla różnych graczy.
- ③ Koszt jest zwykle jednostkowy (inny można uwzględnić w końcowej wypłacie, dodając do stanu „finanse” gracza)

# Drzewo gry



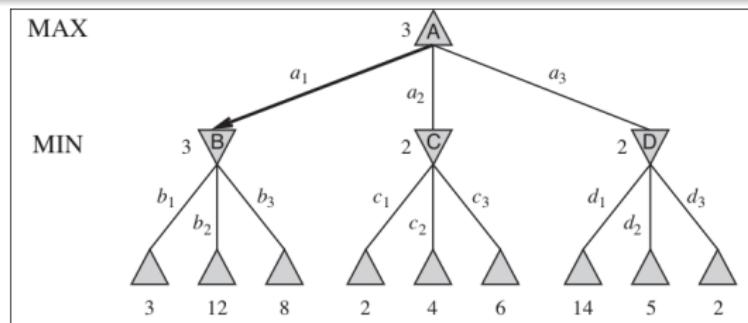
# Kółko i krzyżyk. Drzewo gry

## Fragment drzewa dla prawdziwej gry



## Inna prosta gra (2)

- Mamy dwóch graczy Max i Min (jeden chce maksymalizacji, drugi minimalizacji).
- Wartość dla Max-a to liczba przeciwna wartości dla Min-a.
- Mamy dwa ruchy, zaczyna gracz maksymalizujący.



# Algorytm MiniMax

```
MAX = 1
MIN = 0

def decision(state):
    """decision for MAX"""
    return max(a for actions(state),
               key = lambda a : minmax(result(a,state), MIN))

def minmax(state, player):
    if terminal(state): return utility(state)

    values = [minmax(result(a,state), 1-player) for a in actions(state)]
    if player == MIN:
        return min(values)
    else:
        return max(values)
```

Spotyka się różne warianty nazewnicze (niestety również na naszych slajdach):

- Algorytm MiniMax
- Algorytm min-max
- Algorytm MinMax

- $O(d)$  – pamięć
- $O(b^{2d})$  czas, gdzie  $d$  jest liczbą ply's (półruchów)
- Dla szachów  $b \approx 35$ ,  $d \approx 50$
- Dla go: 250, 150

# Algorytm MiniMax (wersja realistyczna)

- Algorytm MiniMax działa jedynie dla bardzo małych, sztucznych gier (ewentualnie dla końcówek prawdziwych gier).
- Żeby go uczynić realistycznym, musimy:
  - a) Przerwać poszukiwania na jakiejś głębokości.
  - b) Umieć szacować wartość nieterminalnych sytuacji na planszy.

# Algorytm MinMax z głębokością

```
def decision(state):
    return max([a for a in actions(state),
               key = lambda a : minmax(result(a,state), MIN ,0))]

def minmax(state, player, depth):
    if terminal(state): return utility(state)
    if cut_off_test(state, depth):
        return heuristic_value(state)

    values = [minmax(result(a,state), 1-player, depth+1) for a in actions(state)]
    if player == 0:
        return min(values)
    else:
        return max(values)
```

# Dwa parametry algorytmu wyszukiwania

- ① **cut\_off\_test**: kiedy kończymy przeszukiwanie
  - najłatwiej: jak osiągniemy maksymalny poziom, biorąc pod uwagę możliwości
  - Nie jest to jedyne wyjście (ani najlepsze)
- ② Co to znaczy funkcja **heuristic\_value**

# Jak szacować wartość sytuacji?

## Wariant 1

Korzystamy z wiedzy eksperta, próbując ją sformalizować.

## Wariant 2

Próbowemy zaprząć jakiś mechanizm uczenia (lub przeszukiwania), żeby tę funkcję wybrać.

# Jak szacować wartość sytuacji? (2)

Generalne wskazówki:

- ① Przewaga materialna (więcej, lepszych figur)
- ② Ustawienie figur (ruchliwość – liczba możliwych ruchów)
- ③ Szacowana liczba ruchów do zwycięstwa (zagrożony król, itp.).
- ④ Ochrona naszych figur (jak mnie zbijesz, to ja cię zaraz zbiję)

# Aktywny goniec

Biały goniec wprowadzony do gry, czarny nie może nic zrobić.



# Przewaga materialna

- Wartość materialną liczą powszechnie szachiści:
  - a) pion: 1
  - b) skoczek, goniec: 3
  - c) wieża: 5
  - d) hetman: 9
- Sprawdzono doświadczalnie, że te wartości są dobrze dobrane (jak sobie wyobrazić taki eksperyment?)

## Uwaga

Nawet nie wiedząc nic o uczeniu, możemy sobie wyobrazić łatwo jakąś procedurę wyznaczania tych wartości. Na przykład:

1. Losujemy 100 zestawów:  
(1, wartość-gońca, wartość-skoczka, wartość-wieży, wartość-hetmana).
2. Przeprowadzamy pojedynek każdy z każdym.
3. Wybieramy zwycięzcę.



# O grach (część 2)

Paweł Rychlikowski

Instytut Informatyki UWr

8 kwietnia 2021

# Przykładowa gra. Przypomnienie

- Gracz **A**, który chce skończyć z jak największą liczbą, wybiera jeden z trzech zbiorów:
  1.  $\{-50, 50\}$
  2.  $\{1, 3\}$
  3.  $\{-5, 15\}$
- Następnie gracz **B** wybiera liczbę z tego zbioru.

## Składniki

Stany gry (początkowy, 3 po pierwszym ruchu, 6 po drugim), ruchy i mechanika, wypłaty w stanach końcowych.

## Uwaga

W grach o sumie zerowej/stałej

wypłata drugiego gracza = K - wypłata pierwszego (często K=0)

## Connect 4. Inna przykładowa gra



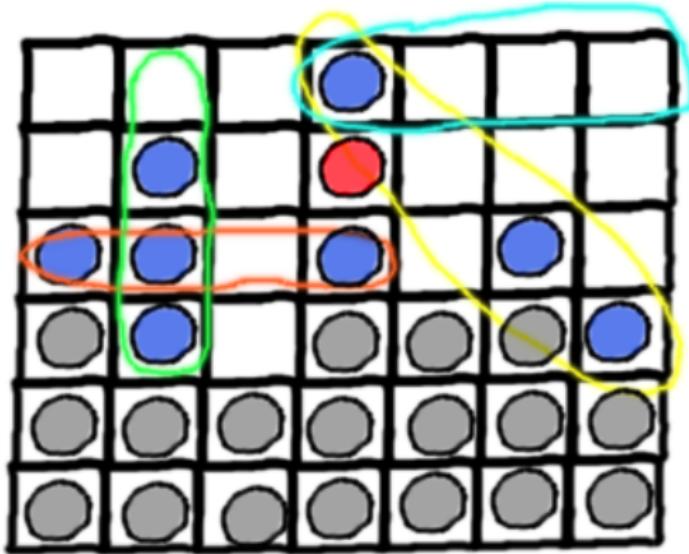
- Prosty, a zarazem grywalny wariant kółka i krzyżyka
- Dodatkowe elementy: mamy ciążenie i piony spadają, gramy do 4 w wierszu, kolumnie lub na przekątnej

# Connect 4. Inna przykładowa gra



- Prosty, a zarazem grywalny wariant kółka i krzyżka
- Dodatkowe elementy: mamy ciążenie i piony spadają, gramy do 4 w wierszu, kolumnie lub na przekątnej

# Co to znaczy wzorzec w Connect 4?



- Analizujemy wszystkie czwórki pól (w każdej bowiem może się zdarzyć układ wygrywający)
- Czwórki, w których są pionki obu kolorów pomijamy
- Wyznaczamy wagę 1-ek, dwójek, trójek (być może zależnie od kierunków)

- Możliwe są większe wzorce, uwzględniające szerszy kontekst
- możliwe jest również **uczenie** większych wzorców. Na przykład za pomocą **splotowych sieci neuronowych (CNN)**.

## Uwaga

Takie sieci działały w AlphaGo.

- Funkcja oceny może być ważoną sumą zaobserwowanych wzorców.
- Wzór:

$$\sum_i w_i p_i$$

( $w_i$  – waga i-tego wzorca,  $p_i$  – ile razy ten wzorzec występuje na planszy)

- Niektóre wagи są dodatnie (mój dobry wzorzec, słabe ustawnienie oponenta), inne ujemne.

# Drobna uwaga o ewolucji. Jak wyznaczyć parametry funkcji oceniającej?

- Istnieje pokusa, żeby zastosować algorytmy ewolucyjne (bo zadanie przypomina ewolucję, w której osobniki toczą ze sobą walkę).
- **Problem:** Jak wyznaczyć funkcję celu?
  - a) Rozgrywać turnieje, przystosowaniem jest średni wynik.
  - b) Wybrać grupę przeciwników (stałą), przystosowaniem X-a będzie średni wynik z tymi przeciwnikami.

## Uwaga

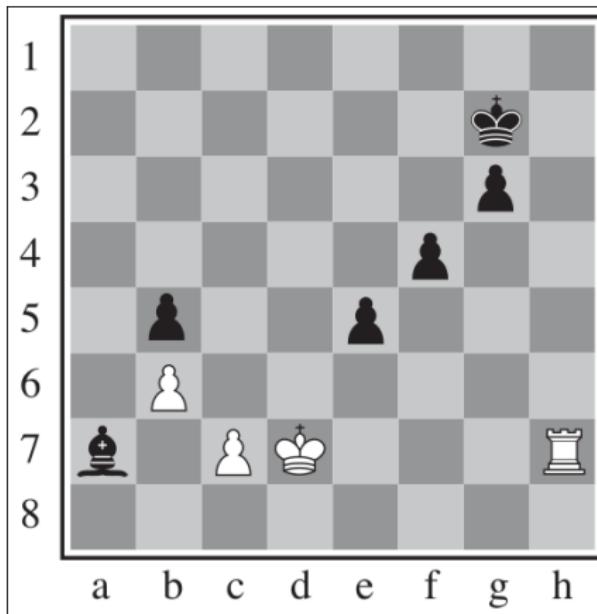
Opcja pełnej ewolucji trochę niebezpieczna, często łączy się oba warianty.

# Przerywanie przeszukiwania

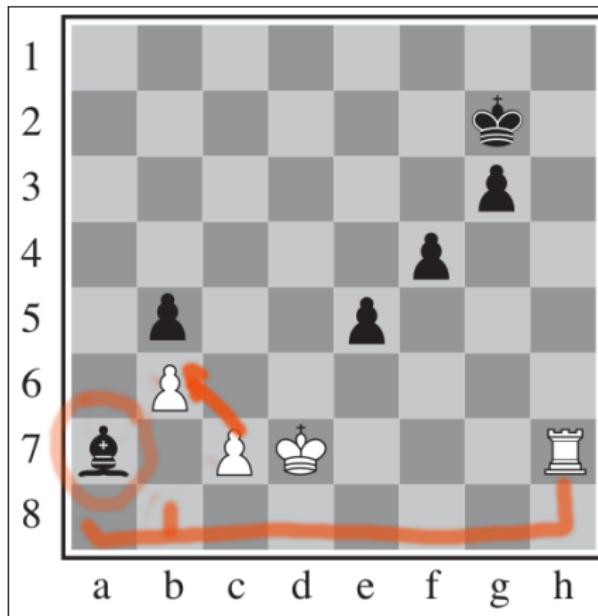
Drugi **metaparametr** funkcji obliczającej wartość planszy.

- Są dwa problemy związane z przerywaniem przeszukiwania:
  - ① Przerwanie w niestabilnej sytuacji (na przykład w środku wymiany hetmanów)
  - ② Tzw. efekt horyzontu (czyli widzimy, że coś się zdarzy, ale w odległej perspektywie)

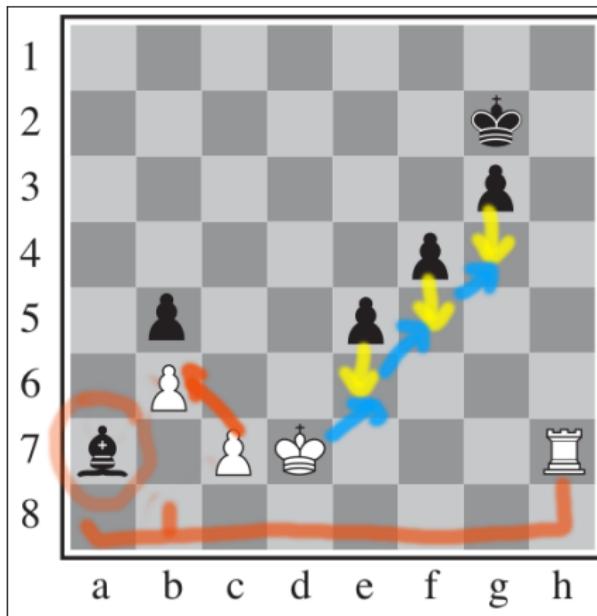
# Efekt horyzontu (zła sytuacja czarnego gońca)



# Efekt horyzontu (zła sytuacja czarnego gońca)



# Efekt horyzontu (zła sytuacja czarnego gońca)



- Nieprzerywanie, jeżeli przeciwnik ma bicie.
- Ogólniej: powyżej jakiejś głębokości rozważamy tylko ruchy mocno zmieniające sytuację

## Definicja

W **przeszukiwaniu z bezruchem** ( quiescence search) możemy skończyć poszukiwanie **tylko** gdy sytuacja jest statyczna.

- Można też stosować jakąś wersję *local beam search* (od któregoś momentu ograniczając mocno rozgałęzienie drzewa)
- Rozważa się warunek **singular extension**, czyli istnienie jednego ruchu, który jest wyraźnie (na oko) lepszy od innych. Takie ruchy zawsze wykonujemy, zwiększając głębokość, a nie zwiększając rozgałęzienia.

## Uwaga

Trochę tak działają ludzie.

# Koniec części I

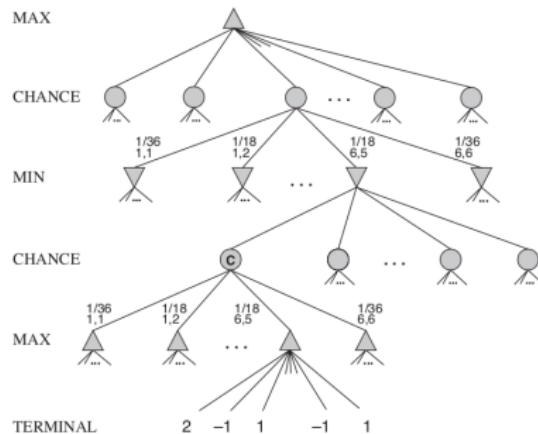
## Inne warianty gier

# Losowość w grach

- W niektórych grach (i w życiu) mamy element losowy.
- Prosty przykład: **szachy z kostką**:
  - Przed ruchem wykonujemy rzut kostką, który determinuje czym możemy się ruszyć,
  - 1 - pionek, 2 - skoczek, 3 - goniec, 4 – wieża, 5 – hetman, 6 – król
  - Gramy do zbicia króla.

# Losowość w grach

- Wprowadzamy dodatkowe węzły, czyli **chance nodes**.
- Przykładowe drzewo gry (dla losowania przy użyciu **dwoch kości**):



# Expectimax

- Minimax, do którego dołożono węzły losowe.
- W węzłach losowych mamy wybór wartości oczekiwanej (sumowanie)

```
def emm(state, player):  
    if terminal(state): return utility(state)  
    if player == MIN:  
        return min( emm(result(state, a), next(player)) for a in actions(state))  
    if player == MAX:  
        return max( emm(result(state, a), next(player)) for a in actions(state))  
    if player == CHANCE:  
        return sum( P(r) * emm(result(state, r), next(player)) for r in actions(state))
```

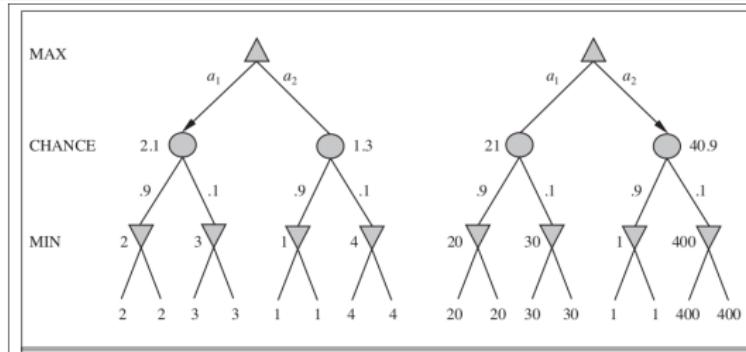
# Wartość sytuacji w grach z losością

## Uwaga 1

Dowolne monotoniczne przekształcenie nie zmienia ruchów wybieranych przez minimax!

## Uwaga 2

W grach z losowością powyższe zdanie przestaje być prawdziwe.



- Analiza gier z losowością jest nieco trudniejsza.
- Mögemy skorzystać z następującej idei:

*Oceniamy sytuację przeprowadzając dużo losowych gier rozpoczęjących się w danej sytuacji*

- Uwaga:** dwa rodzaje losowości: jeden związany z węzłami losowymi (dany przez grę), drugi związany z węzłami min/max – zamiast wyliczać ruch wykonujemy ruch losowy.

## Uwaga

Monte Carlo Simulation dotyczy nie tylko gier z losością!

- Zauważmy, że Monte Carlo Simulation jakoś rozwiązuje problem horyzontu (bo symulacje mogą być b. długie)
- Możemy losować ruchy z niejednakowym prawdopodobieństwem (preferując te, które lokalnie wyglądają sensownie)

## Uwaga

Bardzo ważnym nie tylko w grach jest algorytm **Monte Carlo Tree Search**, o którym jeszcze powiemy.

# Gry częściowo obserwowlne

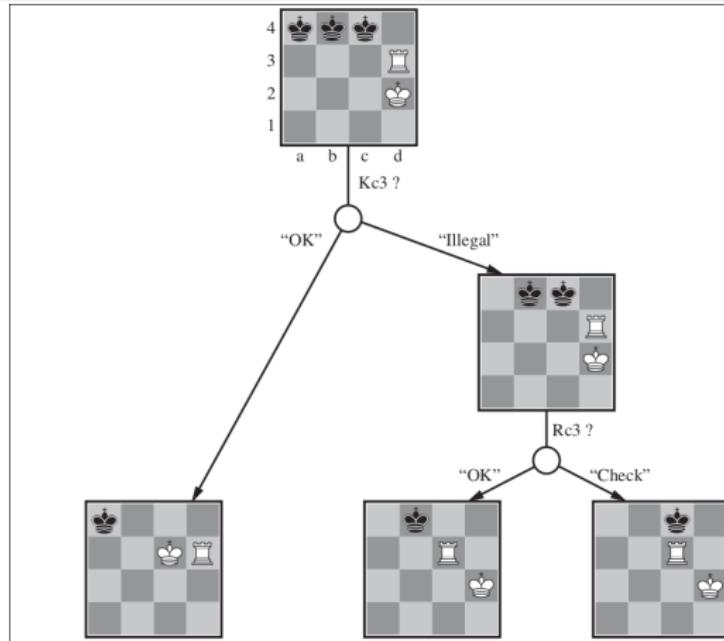
- Ciekawe do analizy są gry, w których agenci nie mają pełnej wiedzy o świecie.
- Klasyczne przykłady to gry karciane, ale nie tylko.

- Mamy dwóch graczy, arbitra i 3 szachownice.
- Gracze widzą na szachownicy swoje pionki, mogą tworzyć też hipotezy o bierkach przeciwnika.
- Arbiter zna położenie wszystkich figur i udziela graczom pewnych (skąpych) informacji.
  - a) przede wszystkim ocenia, czy ruch jest możliwy (komunikacja osobista, dobry ruch jest od razu wykonywany, w przypadku złego, gracz proponuje kolejny, aż do skutku)
  - b) odpowiada na pytanie: „czy ja (gracz) mam jakieś bicie?”
  - c) informuje obu graczy, że „na polu X zbito bierkę” (nie podając jaka bierka jest zbita, a jaka biła)
  - d) Mówi o szachu (do ubu graczy), dodając, że zagrożenie jest w wierszu, kolumnie, przekątnej lub przez skoczka
- Tak poza tym, to całkiem normalne szachy.

Podobno ludzie radzą sobie z tą grą całkiem nieźle...

# Końcówka w Kriegspiel

Przykładowa końcówka, gracz biały dowiedział się, że czarnemu zostało tylko króla i jest on na jednym z 3 pól.



## Uwaga 1

W stanie gry powinniśmy umieścić możliwe ustawienia bierek przeciwnika

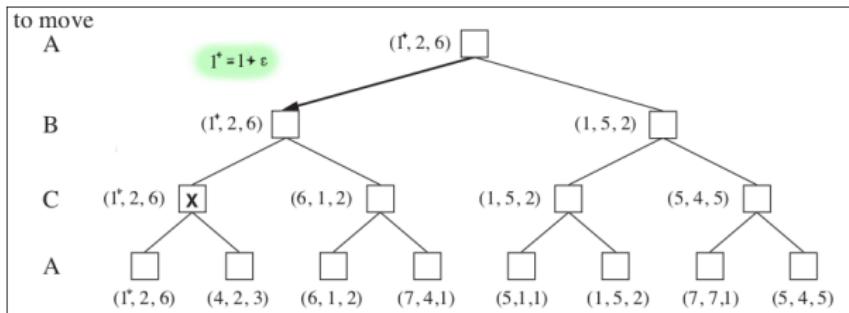
Trochę jak z komandosem...

A jak grać w brydża, bądź inną grę karcianą?

## Idea (do rozwinięcia na ćwiczeniach)

losowanie układu kart i gra w otwarte karty dla wylosowanego układu, czynności powtarzamy wiele razy

# Gry z większą liczbą uczestników



- Strategia maksymalizująca korzyść pojedynczego gracza w oczywisty sposób nieoptimalna ( $A$  mógłby się dogadać z  $B$ ).
- Kwestie sojuszów, zrywania sojuszów, budowania wiarygodności.
- Czasem używa się: **paranoidalnego założenia** – gra wieloosobowa staje się jednoosobową, w której **onи wszyscy** chcą mi zaszkodzić.

Koniec części II

## Uwaga

Początki gier są podobne (bo rozpoczynamy z tego samego stanu startowego)

Z tego wynika, że:

- Możemy np. poświęcić parę godzin, na obliczenie najlepszej odpowiedzi na każdy ruch otwierający.
- Możemy „rozwinąć” początkowy kawałek drzewa (od któregoś momentu tylko dobre odpowiedzi oponenta)
- Możemy skorzystać z literatury dotyczącej początków gry (obrona sycylijska, partia katalońska, obrona bałtycka, i wiele innych)

- Stany mogą się powtarzać (również z zeszłej partii naszego programu).
- Czasem do stanu możemy dojść na wiele sposobów (zwłaszcza, jak ruchy są od siebie niezależne)
- Jeżeli mamy oceniony stan z głębokością 6 i dochodzimy do niego z głębokością 3, to opłaca się wziąć tę bardziej prezencyjną ocenę (w dodatku bez żadnych obliczeń).

## Uwaga

Potrzebny nam jest efektywny sposób pamiętania sytuacji na planszy.

# Tabele transpozycji

- Zapamiętywanie pozycji powinno być efektywne pamięciowo i czasowo.
- Używa się następującego schematu kodowania (**Zobrist hashing**):
  - Mamy zdania typu: **biały goniec jest na g6 (WB-G6)**, **czarny król jest na b4 (BK-B4)**, itd ( $12 \times 64$ )
  - Każde z nich dostaje losowy ciąg bitów (popularny wybór: **64 bity**)
  - Planszę kodujemy jako **xor** wszystkich prawdziwych zdań o tej planszy.
  - Zauważmy, jak łatwo przekształca się te kody:  
nowy-kod = stary-kod **xor** wk-a4 **xor** wk-b5  
to ruch białego króla z a4 na b5

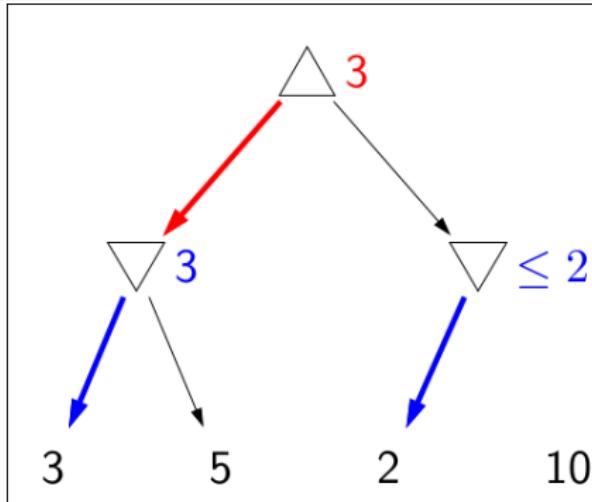
## Uwaga

Często nie przejmujemy się konfliktami, uznając że nie wpływają w znaczący sposób na rozgrywkę.

# Obcinanie fragmentów drzew

## Idea

nie zawsze musimy przeglądać całe drzewo, żeby wybrać optymalną ścieżkę



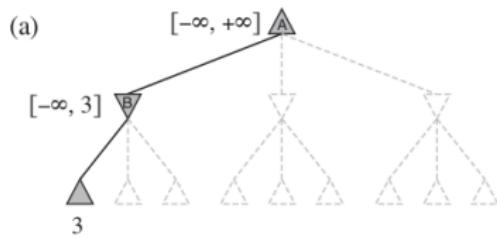
Źródło: CS221, Liang i Ermon

Mamy:  $\max(3, \leq 2) = 3$

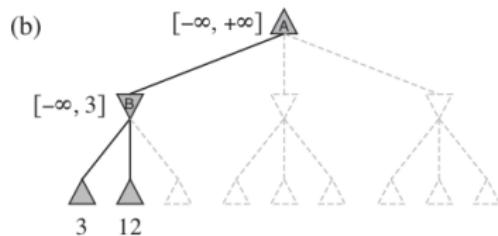
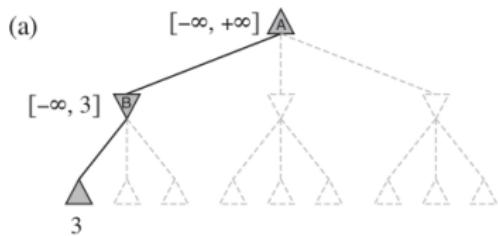
# Obcinanie fragmentów drzew

- Jeżeli możemy udowodnić, że w jakimś poddrzewie nie ma optymalnej wartości, to możemy pominąć to poddrzewo.
- Będziemy pamiętać:
  - $\alpha$  – dolne ograniczenie dla węzłów MAX ( $\geq \alpha$ )
  - $\beta$  – górne ograniczenie dla węzłów MIN ( $\leq \beta$ )

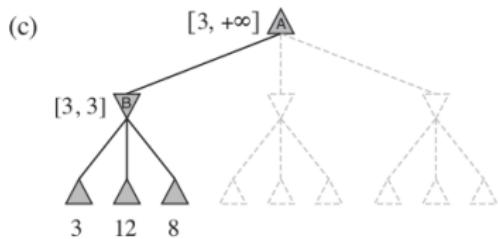
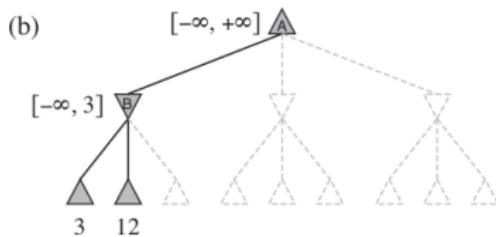
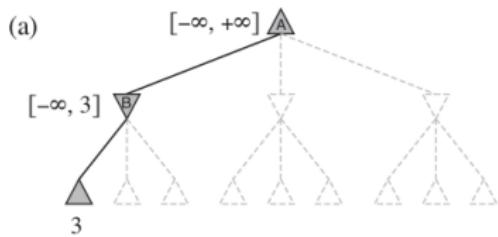
# Alfa-Beta w akcji



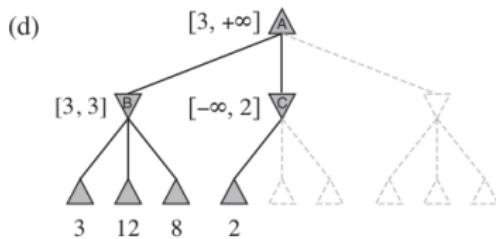
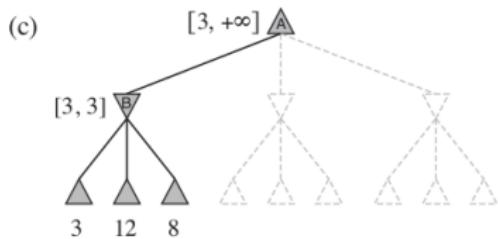
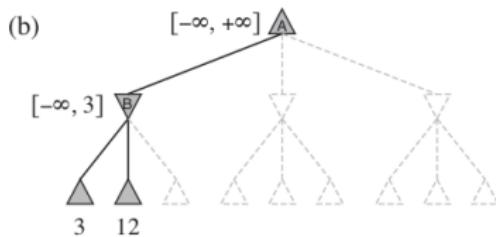
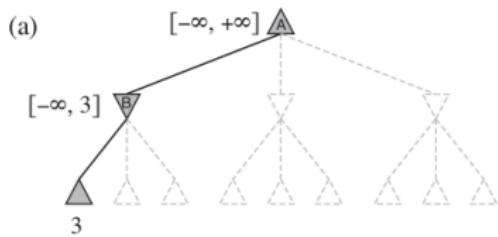
# Alfa-Beta w akcji



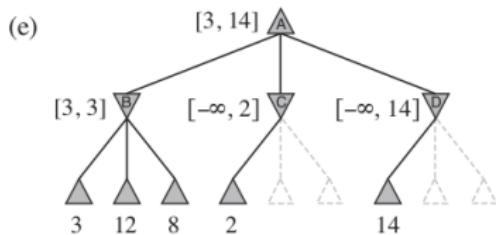
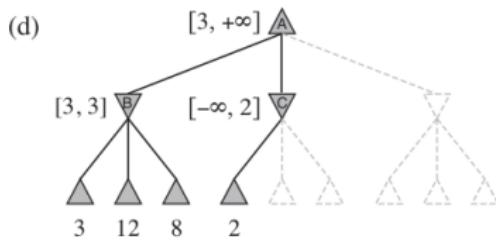
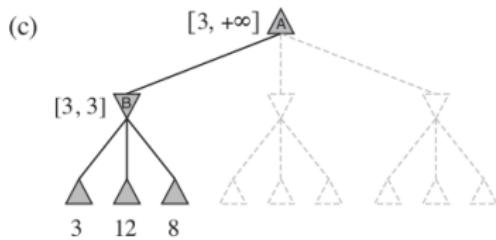
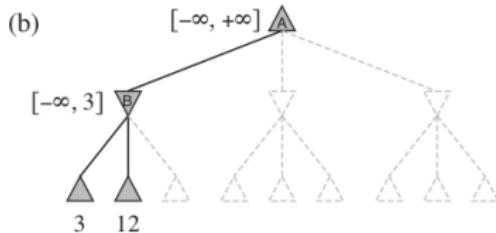
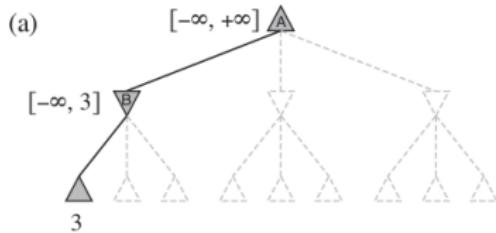
# Alfa-Beta w akcji



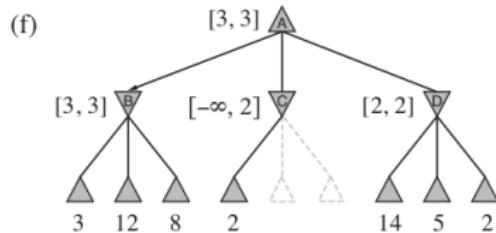
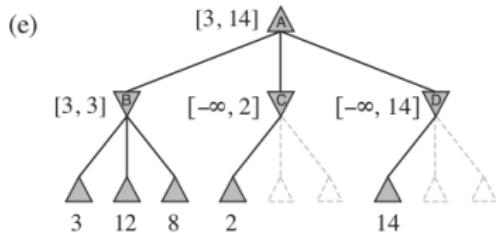
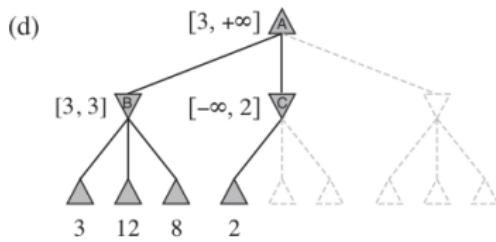
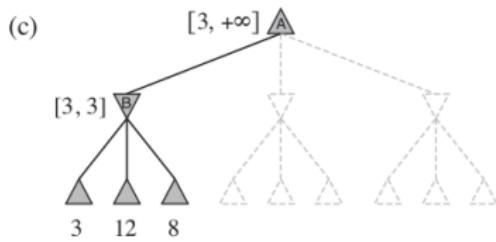
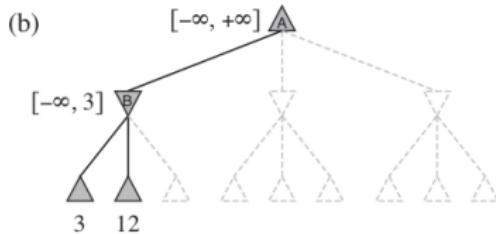
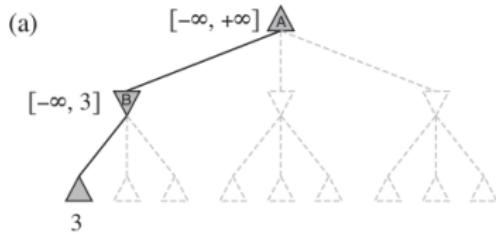
# Alfa-Beta w akcji



# Alfa-Beta w akcji



# Alfa-Beta w akcji



- Będziemy pamiętać:
  - $\alpha$  – dolne ograniczenie dla węzłów MAX ( $\geq \alpha$ )
  - $\beta$  – górne ograniczenie dla węzłów MIN ( $\leq \beta$ )

# Algorytm A-B

```
def max_value(state, alpha, beta):
    if terminal(state): return utility(state)
    value = -infinity

    for state1 in [result(a, state) for a in actions(state)]:
        value = max(value, min_value(state1, alpha, beta))
        if value >= beta:
            return value
        alpha = max(alpha, value)
    return value

def min_value(state, alpha, beta):
    if terminal(state): return utility(state)
    value = infinity

    for state1 in [result(a, state) for a in actions(state)]:
        value = min(value, max_value(state1, alpha, beta))
        if value <= alpha:
            return value
        beta = min(beta, value)

    return value
```

# Kolejność węzłów

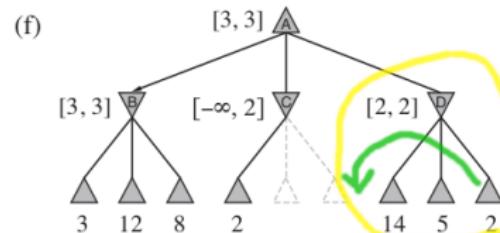
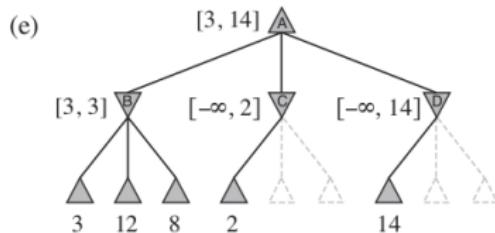
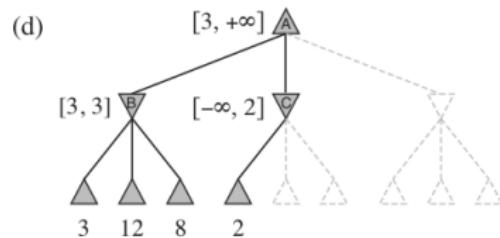
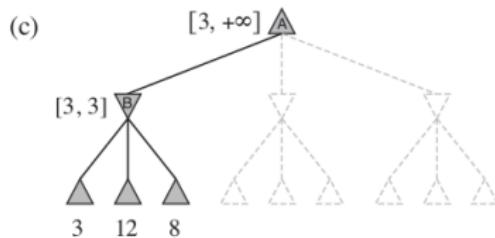
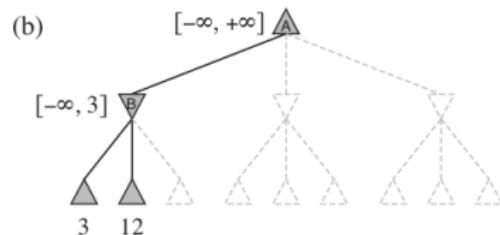
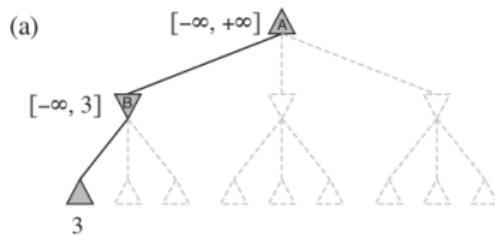
- Efektywność obcięć zależy od porządku węzłów.
- Dla losowej kolejności mamy czas działania  $O(b^{2 \times 0.75d})$  (czyli efektywne zmniejszenie głębokości do  $\frac{3}{4}$ )

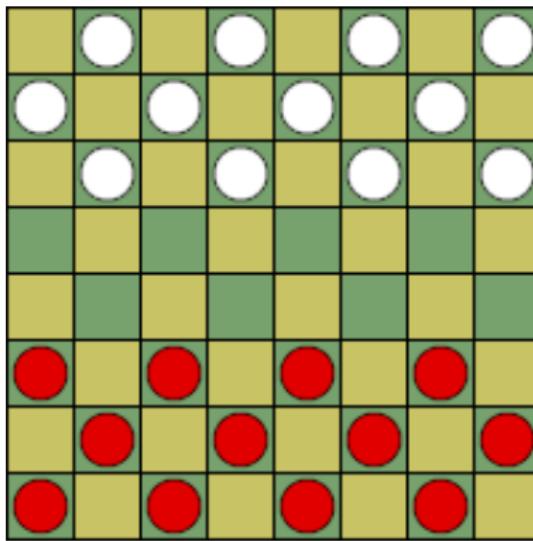
Dobrym wyborem jest użycie funkcji `heuristic_value` do porządkowania węzłów.

## Uwaga

Warto porządkować węzły jedynie na wyższych piętrach drzewa gry!

# Zmiana kolejności wpływa na efektywność





- Ruch po skosie, normalne pionki tylko do przodu.
- Bicie obowiązkowe, można bić więcej niż 1 pionek.  
Wybieramy maksymalne bicie.
- Przemiana w tzw. damkę, która rusza się jak goniec.

# Warcaby – uczenie się gry

- Pierwszy program, który „uczył” się gry, rozgrywając partie samemu ze sobą.
- Autor: Arthur Samuel, 1965

Przyjrzyjmy się ideom wprowadzonym przez Samuela.

- ① Alpha-beta search (po raz pierwszy!) i spamiętywanie pozycji
- ② Przyśpieszanie zwycięstwa i oddalanie porażki: mając do wyboru dwa ruchy o tej samej ocenie:
  - wybieramy ten z dłuższą grą (jeżeli przegrywamy)
  - a ten z krótszą (jeżeli wygrywamy)

# Idea uczenia przez granie samemu ze sobą

## Wariant 1

Patrzymy na pojedynczą sytuację i próbujemy z niej coś wydedukować.

## Wariant 2

Patrzymy na pełną rozgrywkę i:

- a) Jeżeli wygraliśmy, to znaczy, że nasze ruchy były dobre a przeciwnika złe
- b) W przeciwnym przypadku – odwrotnie.

W programie Samuela użyty był wariant pierwszy. Program starał się tak modyfikować parametry funkcji uczącej, żeby możliwie przypominała **minimax** dla głębokości 3 z bardzo prostą funkcją oceniającą (liczącą bierki).

# Symulacje w grach. Podstawy teorii gier

Paweł Rychlikowski

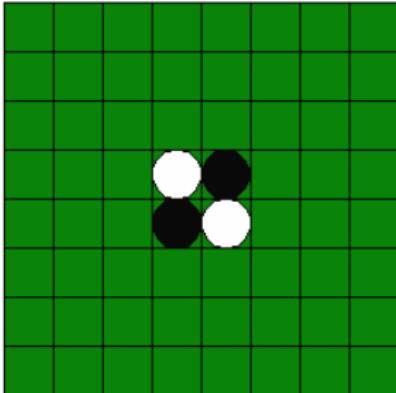
Instytut Informatyki UWr

15 kwietnia 2021

- Gra znana od końca XIX wieku.
- Od około 1970 roku pod nazwą Othello.

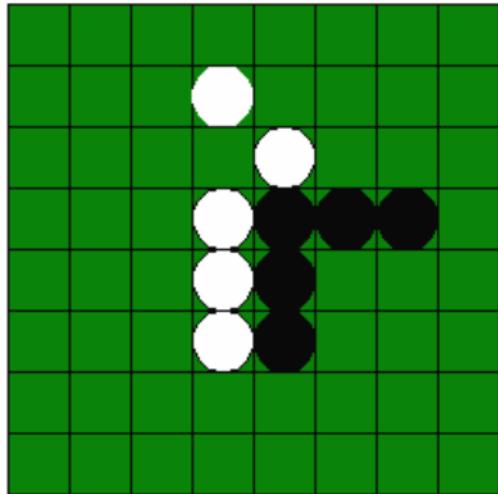
Nadaje się dość dobrze do prezentacji pewnych idei związanych z grami: uczenia i Monte Carlo Tree Search.

# Reversi. Zasady



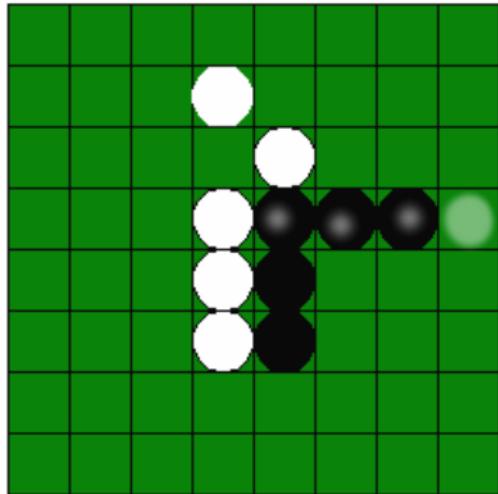
- Zaczynamy od powyższej pozycji.
- Gracze na zmianę dokładają pionki.
- Każdy ruch musi być biciem, czyli okrążeniem pionów przeciwnika w wierszu, kolumnie lub linii diagonalnej.
- Zbite pionki zmieniają kolor (możliwe jest bicie na więcej niż 1 linii).
- Wygrywa ten, kto pod koniec ma więcej pionków.

# Reversi. Ruchy



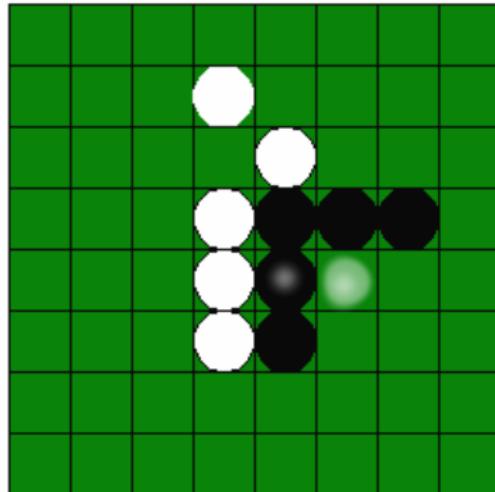
Ruch przypada na białego.

# Reversi. Ruchy

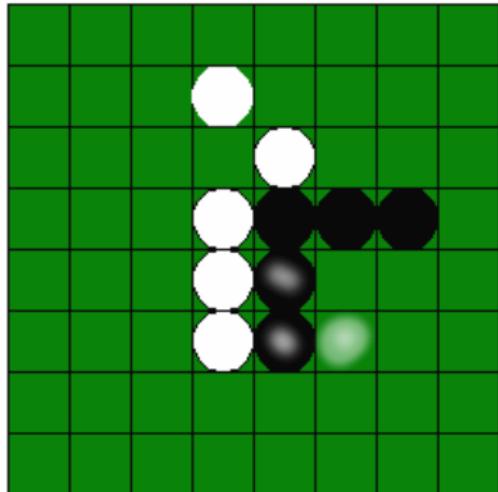


Bicie w poziomie

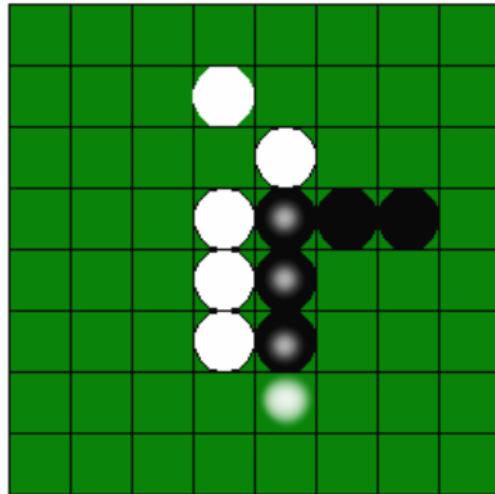
# Reversi. Ruchy



Bicie w poziomie



Bicie w poziomie i po skosie



Bicie w pionie

# Przykładowa gra

- Popatrzmy szybko na przykładową grę.
- **Biały:** minimax, głębokość 3, funkcja oceniająca = balans pionków
- **Czarny:** losowe ruchy

Prezentacja: reversi\_show.py

# Przykładowa gra. Wnioski

## Wniosek 1

Gracz losowy działa całkiem przyzwoicie. Może to świadczyć o sensowności oceny sytuacji za pomocą symulacji.

## Wniosek 2

Jest wyraźna potrzeba **nauczenia się** sensowniejszej funkcji oceniającej.

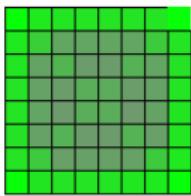
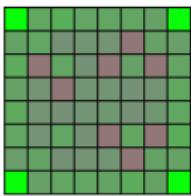
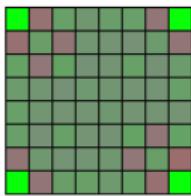
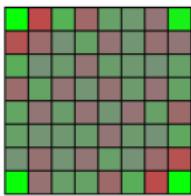
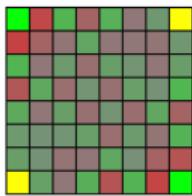
## Cel

Ocena wartości pól w różnych momentach gry (pod koniec wiadomo jaka).

1. Wykonujemy losowych  $K$ -ruchów. Będziemy oceniać wartość pól po  $K$  ruchach.
2. Rozgrywamy partię po tych  $K$  ruchach:
  - a. Białe wygrały: zwiększamy trochę wartość pól zajętych przez białe, zmniejszamy wartość pól zajętych przez czarne.
  - b. Czarne wygrały: postępujemy odwrotnie.

# Wyniki eksperymentu

- Zielone – pozytywne pola, warto na nich mieć pionka w momencie  $K$ .
- Czerwone – tych pól powinniśmy raczej unikać (w momencie  $K$ ), mają bowiem wartość ujemną, czyli utrzymując je zajęte, częściej przegrywamy niż wygrywamy.



Wyniki dla  $K = 6, 10, 30, 40, 56$

# Koniec części I

## Wariant życiowy

Jesteśmy na wakacjach, jemy obiad w restauracji. Nawet smakowało. Powtarzamy, czy szukamy innego miejsca?

- Standardowy dylemat agenta działającego w nieznanym środowisku:
  - ① Maksymalizować swoją korzyść biorąc pod uwagę aktualną wiedzę o świecie.
  - ② Starać się dowiedzieć więcej o świecie, być może ryzykując nieoptymalne ruchy.
- Pierwsza strategia to **eksplotacja**, druga to **eksploracja**.

# Jednoręki bandyta



Źródło: Wikipedia

Po pociągnięciu za rączkę, pojawia się wzorek, który (potencjalnie) oznacza naszą niezeroową wypłatę.

- Mamy wiele tego typu maszyn.
- Możemy zapomnieć o wzorkach, maszyny po prostu generują wypłatę, zgodnie z nieznanym rozkładem.
- Bardzo wyraźnie widać dylemat eksploracja vs eksplotacja.

- **Zachłanna:** każda rączka po razie, a następnie... ta która dała najlepszy wynik.
  - **Lepiej:** najlepszy wynik do tej pory
- **$\varepsilon$ -zachłanna:** rzucamy monetą. Z  $p = \varepsilon$  wykonujemy ruch losową rączką, z  $p = 1 - \varepsilon$  – wykonujemy ruch rączką, która ma najlepszy **średni** wynik do tej pory.
- **Optymistyczna wartość początkowa:** inny sposób na zapewnienie eksploracji. Na początku każdy wybór obniża atrakcyjność danego bandyty.

# Upper Confidence Bound

- Wybieramy akcję  $a$  (bandytę) maksymalizującą:

$$Q_t(a) + c \sqrt{\frac{\ln t}{N_t(a)}}$$

gdzie:  $Q_t$  to uśredniona wartość akcji do momentu  $t$ ,  $N_t$  – ile razy dana akcja była wybierana (do momentu  $t$ )

- Zwróćmy uwagę, że jak akcja nie jest wybierana, to prawy składnik powoli rośnie. Akcja wybierana natomiast traci „premię eksploracyjną”, na początku w szybkim tempie (wzrost mianownika).

## Uwaga

Bardzo powszechnie używana strategia! (np. w AlphaGo)

Algorytm odpowiedzialny za przełom w:

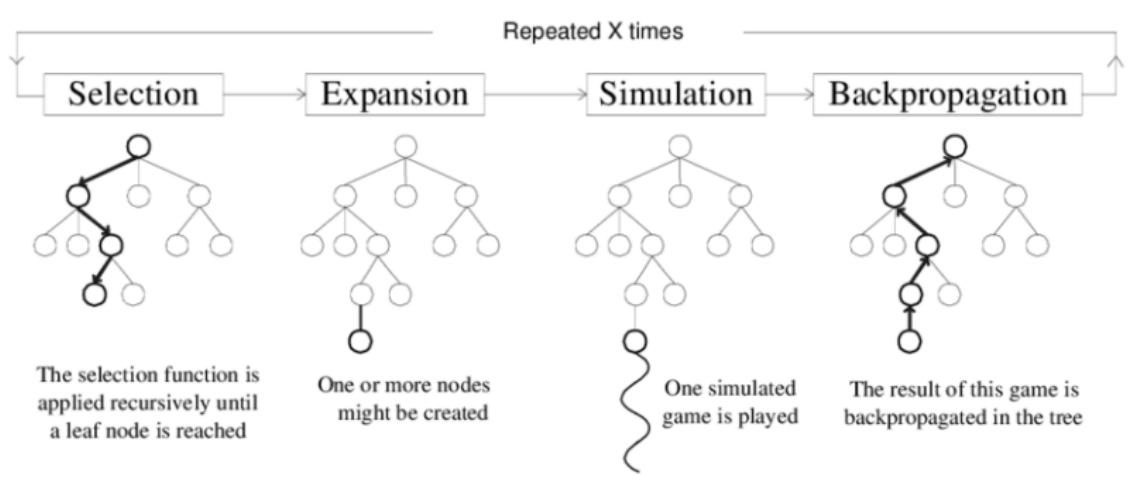
- a. W grze w Go
- b. W General Game Playing

## Główne idee

1. Oceniamy sytuację wykonując symulowane rozgrywki.
2. Budujemy drzewo gry (na początku składające się z jednego węzła – stanu przed ruchem komputera)
3. Dla każdego rozwiniętego węzła utrzymujemy statystyki, mówiące o tym, kto częściej wygrywał gry rozpoczętujące się w tym węźle
4. Selekcję wykonujemy na każdym poziomie (UCB), na końcu rozwijamy wybrany węzeł dodając jego dzieci i przeprowadzając rozgrywkę.

- ① **Selection:** wybór węzła do rozwinięcia
- ② **Expansion:** rozwinięcie węzła (dodanie kolejnych stanów)
- ③ **Simulation:** symulowana rozgrywka (zgodnie z jakąś polityką), zaczynające się od wybranego węzła
- ④ **Backup:** uaktualnienie statystyk dla rozwiniętego węzła i jego przodków

# MCTS. Rysunek



## Inna opcja

**Rozwinięcie** to dodanie wszystkich dzieci i przeprowadzenie dla nich po jednej symulowanej rozgrywce (powyższy rysunek zakłada **rozwinięcie częściowe**, wówczas dochodząc do węzła kolejny raz powinniśmy wziąć kolejny ruch, aż do uzyskania rozwinięcia pełnego).

- Rozgrywka nie musi być prostym losowaniem, p-stwo ruchu może zależeć od jego (**szynkowej!**) oceny.
- Im więcej symulacji, tym lepsza gra – precyzyjne sterowanie trudnością i czasem działania.

## Wybór ruchu

- Naturalny wybór: ruch do najlepiej ocenianej sytuacji
- Inna opcja: ruch do sytuacji, w której byliśmy najwięcej razy

- Rozgrywka nie musi być prostym losowaniem, p-stwo ruchu może zależeć od jego (**szymbkiej!**) oceny.
- Im więcej symulacji, tym lepsza gra – precyzyjne sterowanie trudnością i czasem działania.

## Wybór ruchu

- Naturalny wybór: ruch do najlepiej ocenianej sytuacji
- **Lepsza opcja: ruch do sytuacji, w której byliśmy najwięcej razy**

- W pewnym sensie opcje są podobne: UCB też raczej wybiera dobre ruchy (eksploatacja!)
- Wybierając częstą sytuację, uwzględniamy wiarygodność szacunków
- Pojedyncza bardzo korzystna partia zmienia stosunkowo niewiele

- Ciekawa idea: **all-moves-as-first**: w danej sytuacji na planszy szacujemy jakość ruchów widzianych (w symulacjach, w  $\alpha\beta$ -search też by się dało to zastosować) niezależnie od tego, w którym momencie się zdarzyły
- Motywacja: **w tej sytuacji zawsze jak ruszę hetmanem na B5 to wygrywam**
- Możemy liczyć wartość ruchu jako średni wynik rozgrywki, w której ten ruch był wykonany.
- **Uwaga:** nie  $Q(s, a)$ , ale  $Q(a)$ ! (ta wartość nie zależy od konkretnego momentu, w którym ruch został wykonany)

Więcej szczegółów w pracy S.Gelly, D.Silver, *Monte-Carlo Tree Search and Rapid Action Value Estimation in Computer Go*

- Nie tylko do gier!
- Można stosować do *poważnych* zadań, związanych z przeszukiwaniem (bez oponenta)
  - Na przykład do rozwiązywania więzów (pewnie szczegóły na ćwiczeniach)

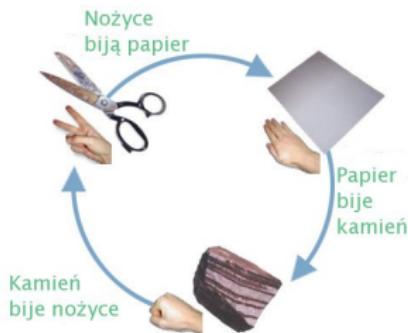
# Koniec części II

# Gry z jedną turą

- Powiemy sobie trochę o grach z jedną turą
- Ale takich, w których gracze podejmują swoje decyzje jednocześnie

Rozważamy gry z **sumą zerową**.

# Papier, nożyce, kamień



Źródło: Wikipedia

## Macierz wypłat

Grę definiuje **macierz wypłat**. Przykładowo poniżej dla P-N-K

Max/Min	Papier	Nożyce	Kamień
Papier	0	-1	+1
Nożyce	+1	0	-1
Kamień	-1	+1	0

- Czysta strategia: zawsze akcja  $a$
- Mieszana strategia: rozkład prawdopodobieństwa na akcjach

- **Oczywisty fakt:** każdą strategię stałą można pokonać (też stałą strategią)
- **Fakt 1:** każdą strategię mieszaną można (prawie) pokonać za pomocą strategii stałej:  
Mój przeciwnik gra losowo, ale z przewagą kamienia – zatem ja daję **zawsze papier**
- **Fakt 2:** Optymalna strategia jest mieszana (w tej grze każde z  $p = \frac{1}{3}$ )
- **Fakt 3:** Znajomość optymalnej strategii mieszanej gracza A, nie daje żadnej przewagi graczowi B (i odwrotnie)

- W prawdziwym P-N-K dochodzi kilka innych aspektów:
  - Grają ludzie, którzy nie potrafią realizować losowości,  
Który człowiek (nie dysponując kostką do gry), przegrawszy 3 razy z rzędu jako papier pokaże papier?
  - za to wysyłają swoimi ciałami różne informacje, które można analizować
- Zatem ma sens organizowanie zawodów w PNK
- Sens miałyby również zawody ludzko-komputerowe, realizowane on-line (agent musiałby zgadnąć, czy gra z człowiekiem, czy z maszyną i czy opłaca się próbować zgadnąć model losowania używany przez człowieka)

# Gra w zgadywanie (Morra 2)

- Mamy dwóch graczy:

- A) Zgadywacz
- B) Zmyłek

którzy na sygnał pokazują 1 lub 2 palce.

- Jeżeli **Zgadywacz** nie zgadnie (pokazał coś innego niż **Zmyłek**), daje **Zmyłkowi** 3 dolary.
- Jeżeli **Zgadywacz** zgadnie, to dostaje od **Zmyłka**:
  - jak pokazali 1 palec, to 2 dolary
  - jak pokazali 2 palce, to 4 dolary

## Pytanie

Jak grać w tę grę? (prośba o podanie wstępnych intuicji)

# Macierz wypłat

## Definicja

Taką grę zadajemy za pomocą **macierzy wypłat**, w której  $V_{a,b}$  jest wynikiem gry z punktu widzenia pierwszego gracza.

Nasza gra:

Zg/Zm	1 palec	2 palce
1 palec	2	-3
2 palce	-3	4

- Jak **Zmyłek** będzie grał cały czas to samo, to **Zgadywacz** wygra każdą turę (i odwrotnie)
- Muszą zatem stosować strategie mieszane, ale jakie?

## Definicja

Wartość gry dla dwóch strategii graczy jest równa:

$$V(\pi_A, \pi_B) = \sum_{a,b} \pi_A(a)\pi_B(b)V(a, b)$$

Przykładowo: Zgadywacz zawsze zgaduje 1, Zmyłek wybiera akcję losowo z prawdopodobieństwem 0.5.

**Wynik:**  $-\frac{1}{2}$  (tak samo często zyskuje 2 jak traci 3 dolary)

## Uwaga

Jeżeli gracz  $A$  zapowie, że będzie grał strategią mieszana (i ją poda), wówczas gracz  $B$  może grać strategią czystą (i osiągnie optymalny wynik).

Dlaczego?

## Odpowiedź

- Możemy dla każdej akcji policzyć wartość oczekiwana wypłaty
- i wybrać (dowolną) najlepszą akcję
- (Jeżeli takich akcji jest więcej, wówczas można też dowolnie losować między nimi)

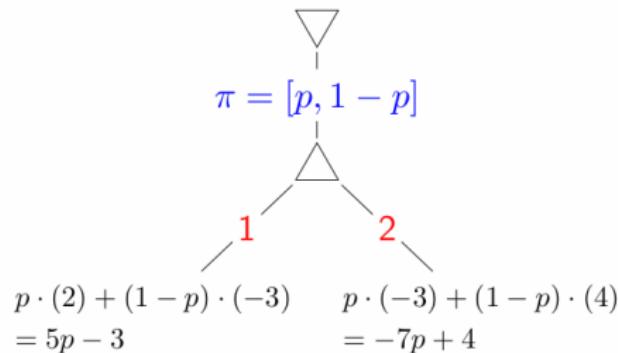
# Gra w zgadywanie (Morra 2). Przypomnienie

- Mamy dwóch graczy:
  - A) Zgadywacz
  - B) Zmyłek
- którzy na sygnał pokazują 1 lub 2 palce.
- Jeżeli **Zgadywacz** nie zgadnie (pokazał coś innego niż **Zmyłek**), daje **Zmyłkowi** 3 dolary.
- Jeżeli **Zgadywacz** zgadnie, to dostaje od **Zmyłka**:
  - jak pokazali 1 palec, to 2 dolary
  - jak pokazali 2 palce, to 4 dolary

# Znalezienie optymalnej strategii

Zaczyna gracz B – Zmyłek.

Wybiera strategię mieszzaną z parametrem  $p$

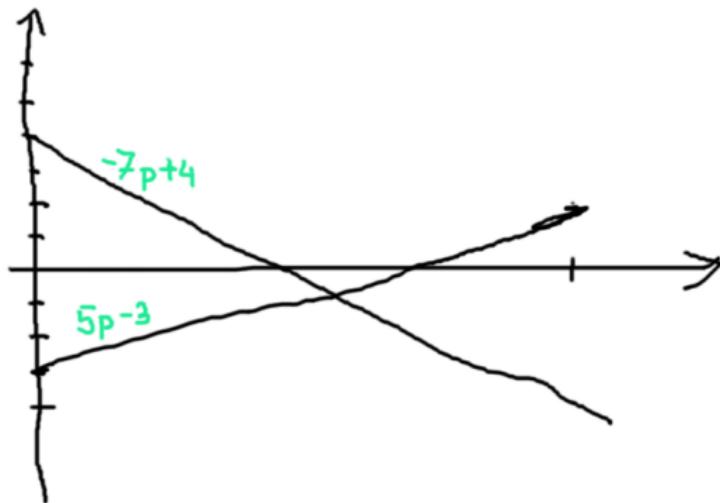


Wartość takiej gry to

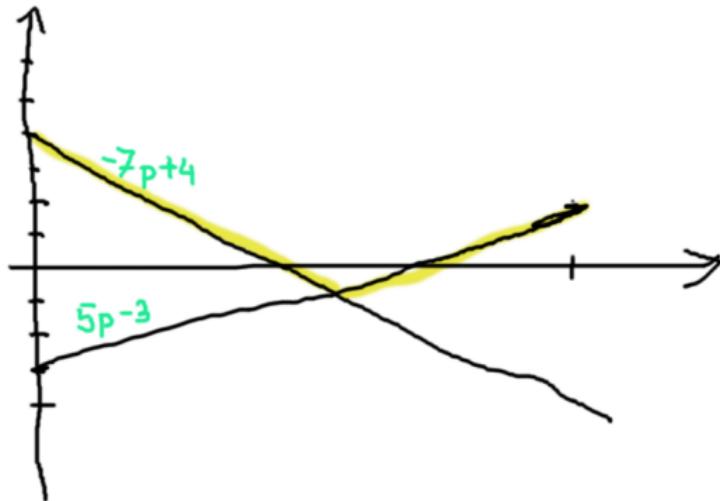
$$\min_{p \in [0,1]} (\max(5p - 3, -7p + 4))$$

Zauważmy, dla jakich  $p$  wygrywa lewe, dla jakich prawe i co z tego wynika.

# Optymalna strategia. Wykresy



# Optymalna strategia. Wykresy



- W powyższej grze, Zmyłek osiągnie najlepszy wynik, gdy przyjmie  $p = \frac{7}{12}$ , wynik ten to  $-\frac{1}{12}$ !
- Ok, on zaczynał, miał trudniej – a gdyby zaczynał Zgadywacz? I podał swoją strategię mieszaną?

## Wynik gry

Wynik jest dokładnie taki sam, czyli  $-\frac{1}{12}$ !

## Twierdzenie, von Neuman, 1928

Dla każdej jednoczesnej gry dwuosobowej o sumie zerowej ze skończoną liczbą akcji mamy:

$$\max_{\pi_A} \min_{\pi_B} V(\pi_A, \pi_B) = \min_{\pi_B} \max_{\pi_A} V(\pi_A, \pi_B)$$

dla dowolnych mieszanych polityk  $\pi_A, \pi_B$ .

- Można ujawnić swoją politykę optymalną!
- **Dowód:** pomijamy, programowanie liniowe, przedmiot J.B.
- Algorytm: programowanie liniowe

- Można o grze wieloturowej myśleć jako o grze jednoturowej
- Gracze na sygnał kładą przed sobą opis strategii (program)

## Uwaga

Optymalną strategią jest MiniMax (ExpectMiniMax w grach losowych). Ale wiedząc o strategii gracza różnej od optymalnej możemy oczywiście ugrać więcej.

- Gry o sumie niezerowej, w których dochodzi możliwość kooperacji.
- Punkt równowagi Nasha (jest zawsze para strategii, że żaden gracz nie chce jej zmienić, wiedząc, że ten drugi nie zmienia).  
**Również dla gier o sumie niezerowej!**
- Agent musi zdecydować, czy ma być miły dla innego agenta (i budować reputację przy wielu rozgrywkach, słynny **dylemat więźnia**).

# Koniec części I

# Procesy decyzyjne Markowa

Paweł Rychlikowski

Instytut Informatyki UWr

22 kwietnia 2021

# Procesy decyzyjne Markowa (MDP)

- Coś pomiędzy grami a zwykłym zadaniem przeszukiwania
- (zwłaszcza jeżeli przypomnijmy sobie gry z węzłami losowymi)
- a jednocześnie krok w stronę uczenia ze wzmacnieniem

# MDP a przeszukiwanie

## Standardowe przeszukiwanie

Znamy mechanikę świata i wiemy, że **akcja w stanie** da nam **konkretny rezultat (inny stan)**.

## MDP

Znamy mechanikę świata i wiemy, że **akcja w stanie** da nam **pewien rozkład prawdopodobieństwa na następnych stanach**.

Nie wiemy, co dokładnie się stanie, ale wiemy co **może** się stać i z jakim prawdopodobieństwem.

- Przyszłość zależy od ostatniego stanu.
- Nie zależy od historii...
- Chyba, że jej fragment (o długości  $N$ ) uznamy za część stanu.

## Ważna uwaga

Zakładamy **skończoną** liczbę stanów

# Uwaga na wulkany (1)

- Dobrze omawia się MDP na prostych światach na prostokątnej kratce.
- I od takich modeli zaczniemy.

Generalnie myślimy na początku o przestrzeni stanów na tyle małej, że nie będzie kłopotów z pamiętaniem różnych wartości dla **każdego stanu**.

# Uwaga na wulkany (2)

## Volcano crossing



		-50	20
		-50	
2			

# Mechanika świata wulkanów

		-50	20
		-50	
2			

- Możliwe 4 akcje (UDLR)
- W normalnym przypadku efekt oczywisty (próba wyjścia poza planszę oznacza pozostanie na polu)
- Z prawdopodobieństwem  $p$  możemy się poślizgnąć, wówczas poruszamy się w losowym kierunku.
- Dojście do pola z liczbą kończy grę (i odpowiednią dostajemy wypłatę).

# Inny przykład. Gra w kości

## Uwaga

Nagroda może być przydzielana w sposób ciągły, nie tylko w stanie końcowym.

- Mamy dwie opcje: **pozostanie** albo **rezygnacja**.
- **rezygnacja** oznacza wypłatę **10\$**
- **pozostanie** to wypłata **4\$** po której rzucamy kostką.
- Interpretacja wyniku:
  - 1,2 – koniec gry
  - 3,4,5,6 – gramy dalej

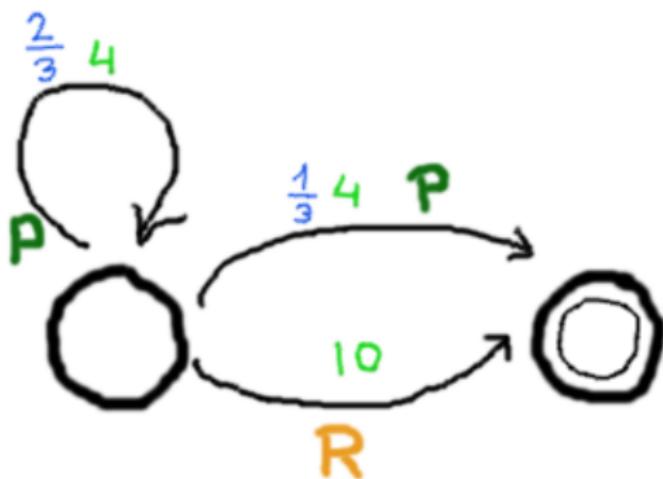
## Pytanie

Ile mamy stanów? Odpowiedź: 2

# Dla gry w kości

- 1 stan z decyzją, dwie **polityki** (czyli sensowne sposoby gry) (schemat na kolejnym slajdzie).
- Możemy policzyć oczekiwanaą wartość dla każdej:
  - **rezygnacja** – 10
  - **pozostanie** – (na kolejnym slajdzie)

# Schemat stanów



Oceniamy strategię **pozostanie**, czyli przedłużania gry.

- Oznaczamy przez  $V$  wartość tej polityki (czyli ile średnio zarobimy, jak nie będziemy nigdy rezygnować z gry)
- $V$  spełnia równanie:

$$V = \frac{2}{3} \times (4 + V) + \frac{1}{3} \times 4 = 4 + \frac{2}{3} \times V$$

- Czyli  $V = 12$ , zatem opłaca się pozostawać w grze (bo  $12 \geq 10$ )

## Uwaga

Tu była tylko jedna decyzja: 10 czy  $V$ , ale podobnie można rozwiązywać również (dużo) bardziej złożone MDP: rozwiązując równania i znajdując wartości stanów.

## Definicja

**Markowski proces decyzyjny** (MDP) zawiera następujące składowe:

1.  $S$  – (skończony) zbiór stanów
2. Stan startowy,  $s_{\text{start}} \in S$
3.  $\text{Actions}(s)$  – zbiór możliwych akcji w stanie  $s$
4.  $T(s,a,s')$  – prawdopodobieństwo przejścia z  $s$  do  $s'$  w wyniku akcji  $a$
5.  $\text{Reward}(s,a,s')$  – nagroda (wypłata) związana z tym przejściem
6.  $\text{IsEnd}(s)$  – czy stan jest końcowy?
7. Discount factor,  $0 < \gamma \leq 1$  – sprawia, że nagrody w przyszłości cieszą mniej.

- Można też myśleć, że dla pary  $(s, a)$  mamy rozkład prawdopodobieństw po parach (nowy-stan, nagroda).
- Nagroda może być pozytywna bądź negatywna

## Uwaga

Oczywiście MDP jest ogólniejsze niż zadanie przeszukiwania (bo wystarczy przypisać niektórym rezultatom p-stwo 1, reszcie 0 i mamy zwykłe zadanie przeszukiwania)

# Czym jest rozwiązańe MDP?

- Przypominamy: rozwiązańiem zadania przeszukiwania jest ciąg akcji (ale to nie tu nie działa, bo?)
  - (wyniki akcji są niedeterministyczne, więc nie wystarczy podać jednego ciągu akcji)
- Rozwiązańie: agent musi wiedzieć, co zrobić w każdym stanie.

## Definicja 1

Polityką deterministyczną nazwiemy funkcję, która każdemu stanowi przypisuje akcję (możliwą w tym stanie).

## Definicja 2

Polityką nazwiemy funkcję, która każdemu stanowi przypisuje rozkład prawdopodobieństwa na akcjach (możliwych w tym stanie).

Koniec części II

## Definicja 1

Polityką deterministyczną nazwiemy funkcję, która każdemu stanowi przypisuje akcję (możliwą w tym stanie).

## Definicja 2

Polityką nazwiemy funkcję, która każdemu stanowi przypisuje rozkład prawdopodobieństwa na akcjach (możliwych w tym stanie).

- Gdy używamy **polityki**, otrzymujemy ciąg stanów, akcji i nagród
- Dla takiej ścieżki możemy zsumować nagrody, otrzymując **użyteczność** dla tej ścieżki
- **Wartością** polityki jest oczekiwana użyteczność polityki (tzn. wartość oczekiwana zmiennej losowej wyrażającej użyteczność takiej ścieżki)

- Realizując politykę, otrzymaliśmy ciąg stanów, nagród i akcji
  - $s_0, a_1, r_1, s_1, a_2, r_2, s_2 \dots, s_n, a_{n+1}, r_{n+1}, s_{n+1} \dots$
- Nagroda po uwzględnieniu zniżek:

$$r_1 + \gamma r_2 + \gamma^2 r_3 + \gamma^3 r_4 + \dots$$

- Widzimy że:
  - Dla  $\gamma = 1$  po prostu sumujemy nagrody cząstkowe
  - Dla  $0 < \gamma < 1$  mamy możliwość mówienia o wartości nieskończonych ciągów akcji.

- Zwróćmy uwagę, że **discounting** ma sens również w przypadku, gdy nagroda wypłacana jest **jedynie** w stanie końcowym.
- Jeżeli wypłata jest tylko w ostatnim stanie, to:
  - a) Agent, który **wygrywa** ( $R > 0$ ) woli dostać ją wcześniej,
  - b) agent, który **przegrywa** ( $R < 0$ ) woli dostać ją później.

Przyśpieszanie zwycięstwa i opóźnianie porażki jest „sensownym” zachowaniem.

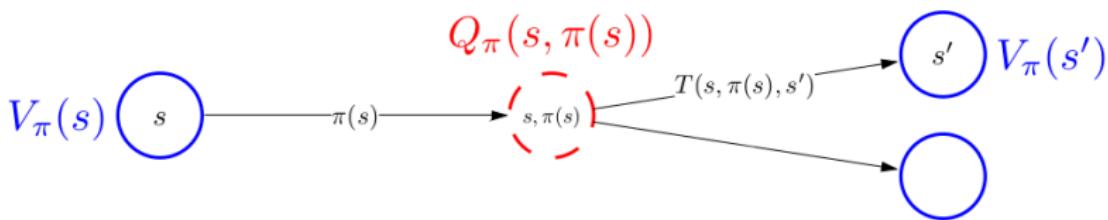
# Wartość polityki

## Definicja

Wartość  $V_\pi(s)$  jest oczekiwana użytecznością dla agenta startującego w stanie  $s$  i działającego zgodnie z polityką  $\pi$

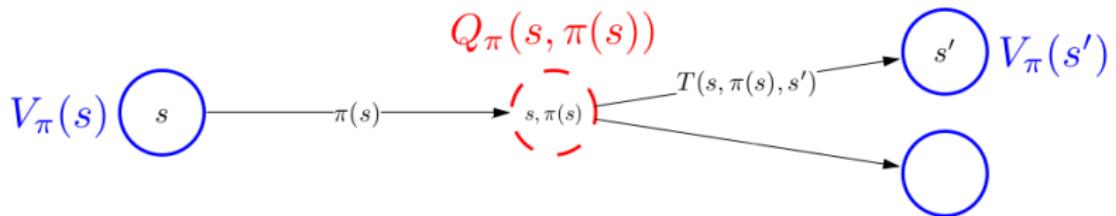
## Definicja

Wartość  $Q_\pi(s, a)$  jest oczekiwana użytecznością dla agenta startującego w stanie  $s$ , wykonującego w tym stanie akcję  $a$  i **dalej** działającego zgodnie z polityką  $\pi$



Źródło: CS221 / Autumn 2017 / Liang & Ermon

# Zależności pomiędzy V i Q



$$Q_\pi(s, a) = \sum_{s'} T(s, a, s') (\text{Reward}(s, a, s') + \gamma V_\pi(s'))$$

# Algorytm: policy evaluation

- Napiszmy rekurencyjny wzór dla wartości  $V$  (przy zadanej polityce)
- 

$$V_{\pi}(s) = \sum_{s'} T(s, \pi(s), s') (\text{Reward}(s, \pi(s), s') + \gamma V_{\pi}(s'))$$

- Mamy układ równań (liniowych), który można rozwiązywać standardowymi metodami.
- Równań jest tyle co stanów (czyli potencjalnie sporo)

Moglibyśmy ten wzór zmodyfikować, mówiąc: zamiast nieznanego  $V$  po prawej stronie weźmiemy poprzednie przybliżenie  $V$ :

$$V_{\pi}^{(t+1)}(s) = \sum_{s'} T(s, \pi(s), s') (\text{Reward}(s, \pi(s), s') + \gamma V_{\pi}^{(t)}(s'))$$

# Algorytm: policy evaluation (3)

- 1 Zainicjuj  $V_{\pi}^{(0)}(s) \leftarrow 0$ , dla wszystkich  $s$
- 2 Powtarzaj dla  $t = 1, \dots, t_{PE}$ 
  - Powtarzaj dla każdego stanu  $s$

$$V_{\pi}^{(t+1)}(s) \leftarrow \sum_{s'} T(s, \pi(s), s') (\text{Reward}(s, \pi(s), s') + \gamma V_{\pi}^{(t)}(s'))$$

- Kończymy, gdy dla każdego stanu zmiana mniejsza niż  $\varepsilon$
- Oczywiście nie musimy pamiętać całej historii, tylko dwa ostatnie jej elementy (stany zmieniane i poprzednie)

## Złożoność

$O(t_{PE}SS')$ , gdzie  $S$  to liczba stanów, a  $S'$  (maksymalna) liczba stanów z niezerową  $T(s, a, s')$ .

- Interesuje nas wyznaczanie polityki (a nie tylko ocenianie jej wartości).

## Definicja

Optymalną wartością stanu  $V_{opt}(s)$  jest maksymalna wartość stanu (ze względu na wszystkie polityki).

# Rekurencja dla polityki optymalnej

Jaka polityka jest optymalna? Taka, która wybiera stany o optymalnej wartości

- Przypominamy, dla każdej polityki mamy:

$$Q_{\pi}(s, a) = \sum_{s'} T(s, a, s')(\text{Reward}(s, a, s') + \gamma V_{\pi}(s'))$$

- Dla polityki optymalnej:  $V_{\text{opt}}(s) = \max_{a \in \text{Actions}(s)} Q_{\text{opt}}(s, a)$

Mögemy podstawić do drugiego wzoru wzór na  $Q_{\pi}$  dla  $\pi = \text{opt}$ .

$$V_{\text{opt}}(s) = \max_{a \in \text{Actions}(s)} \sum_{s'} T(s, a, s')(\text{Reward}(s, a, s') + \gamma V_{\text{opt}}(s'))$$

## Polityka optymalna (do poprzedniego slajdu)

$$\pi_{\text{opt}}(s) = \arg \max_{a \in \text{Actions}(s)} Q_{\text{opt}}(s, a)$$

Nasz wzorek zmieniony na wersję do iterowania

$$V_{\text{opt}}^{(t+1)}(s) = \max_{a \in \text{Actions}(s)} \sum_{s'} T(s, a, s') (\text{Reward}(s, a, s') + \gamma V_{\text{opt}}^{(t)}(s'))$$

## Algorytm Bellmana (value iteration)

- Mamy dodatkową pętlę wybierającą optymalną akcję (zamiast akcji danej przez politykę)
- Reszta bez zmian, tak jak w **policy evaluation**.

# Warunki zbieżności

Algorytm jest zbieżny, jeżeli zachodzi któryś z warunków

- $\gamma < 1$
- Graf MDP jest acykliczny

## Uwaga

W tym ostatnim przypadku wymagana jest jedna iteracja, w której stany przeglądane są w odwrotnym porządku topologicznym (wyjaśnienie na ćwiczeniach)

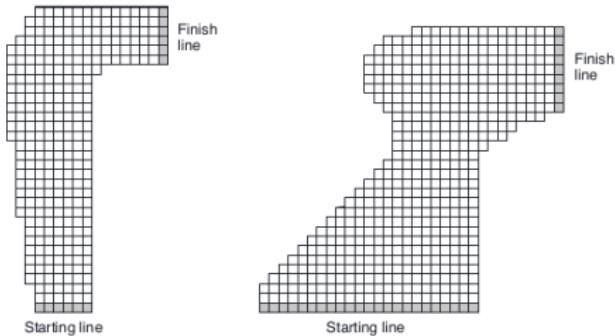
## Uwaga

Zwróćmy uwagę na to co się dzieje, jeżeli  $\gamma = 1$  i mamy cykl.

Dla niezerowych nagród na krawędziach cyklu wartość oczekiwana może być nieokreślona

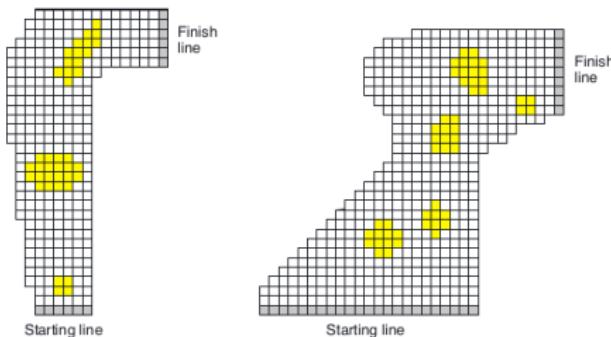
Koniec części II

# Przykład. Wyścigi samochodzików.



- Prędkość autka jest wektorem  
 $(dx, dy) \in \{-3, -2, \dots, 2, 3\} \times \{-3, -2, \dots, 2, 3\}$
- Akcja: zmiana prędkości (każda składowa o co najwyżej 1)
- Celem jest (przejechać) przez metę (możemy to uprościć poszerzając metę i mówiąc, że celem jest dotarcie do piksela mety)
- W pełni deterministyczny świat (BFS, A\*?)

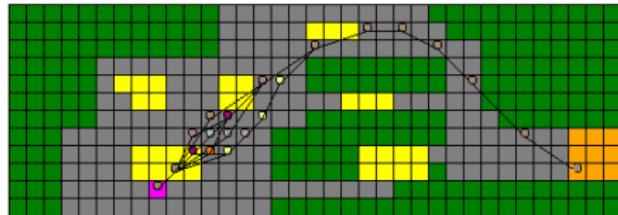
## Wyścigi samochodzików. (2)



- Dodajemy plamy po oleju
- Ruch z pola oleju dodaje dodatkową składową losową do prędkości (znamy rozkład).

W tym momencie klasyczne MDP + algorytm Bellmana (czyli iteracji wartości) powinny dać dobry wynik.

# Wynik algorytmu Value Iteration



Zwróćmy uwagę, że bez żadnych dodatkowych obliczeń można umieszczać w innych miejscach punkt startowy.

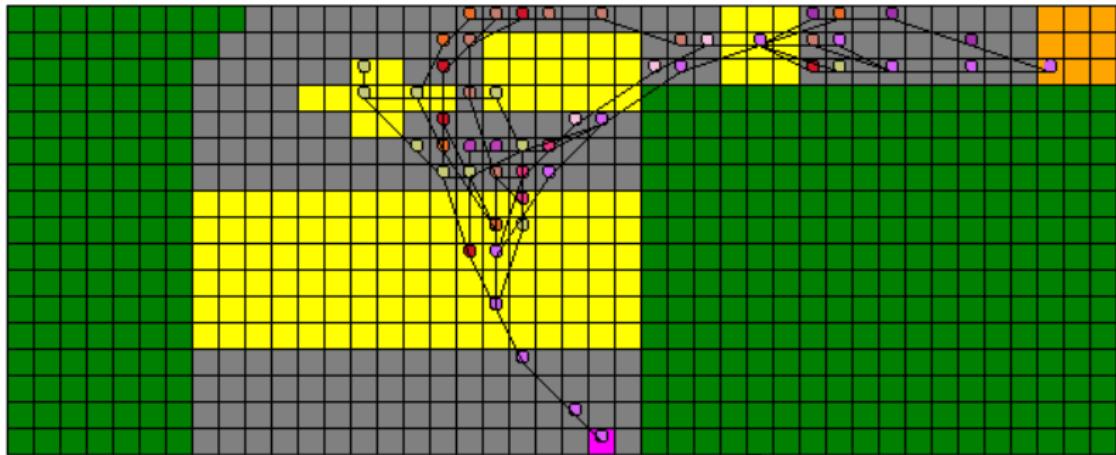
- Fajnie jest dojechać na metę. (+100)
- Ale jeszcze fajniej nie dać się zabić. (-100?)

## Uwaga

Pamiętamy, że monotoniczna zmiana funkcji wypłaty:

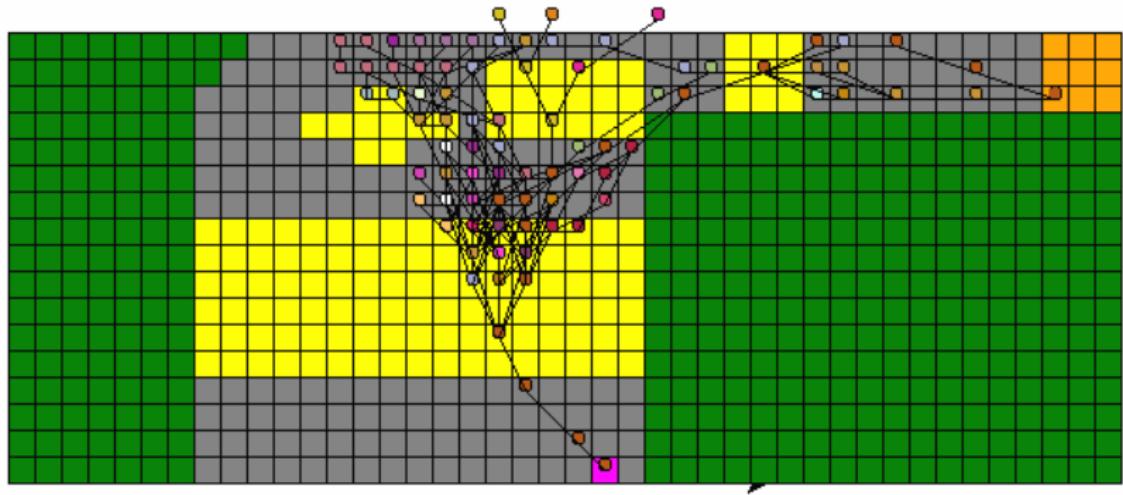
- ① nie zmienia wartości MiniMax-owej gry,
- ② może zmienić ExpectMinMax

# Rozwiążanie podstawowe

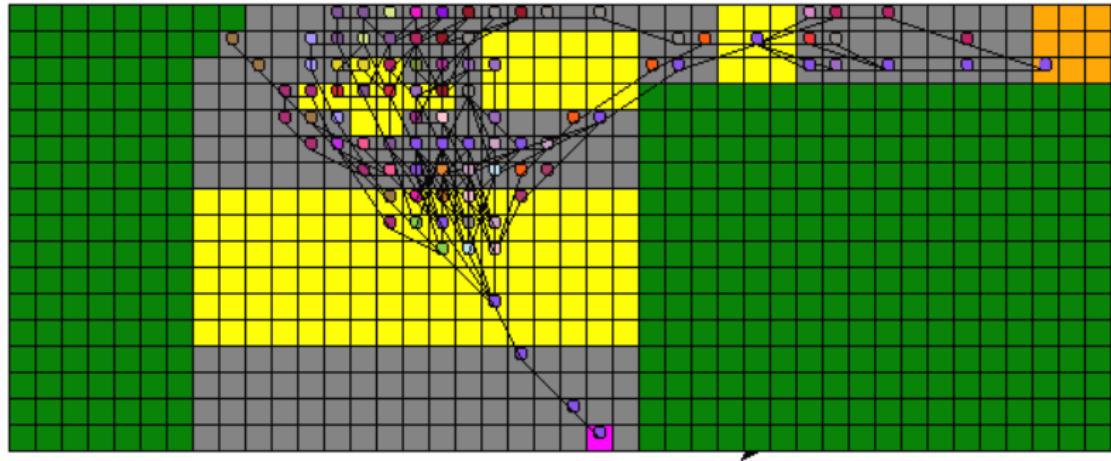


**Pytanie:** Czego spodziewamy się, jeżeli zamienimy karę na wypadek na **10000**?

# Kara=100



# Kara=10000



- Problem: dużo większa plansza, dużo większa liczba stanów.
- **Pomysł 1:** położenie „rozmyte”, na przykład w kwadracie  $10 \times 10$  pikseli.
- **Pomysł 2:** dodatkowo informacja, czy jestem 1, 2, czy 3 raz w takim kwadracie ( $3 < 100$ )

**Fundamentalny problem:** nie znamy mechaniki takiego świata (i wielu innych)

# Wyścigi samochodzików. Float

- Prędkość autka jest wektorem  $(v \cos(d), v \sin(d))$ ,
- Możemy zmieniać  $d$  (skręcać), oraz  $v$  (przyśpieszać, hamować)
- Celem jest meta.
- W pełni deterministyczny świat, ale **bardzo duża liczba stanów, zawierających liczby float**)

- Możemy stworzyć **stan abstrakcyjny** i opisać mechanikę świata dla takich stanów
- Oczywiście będzie ona niedeterministyczna, bo nigdy nie będziemy wiedzieć, czy zmiana w świecie float-ów przenosi się na zmianę w świecie int-ów.

## Uwaga

Możemy myśleć o tym, że modelujemy błędy pomiarowe (int zamiast float) za pomocą losowości.

# Uczenie maszynowe

Paweł Rychlikowski

Instytut Informatyki UW

30 kwietnia 2021

- Zakładamy, że nie dysponujemy modelem (czyli przejściami, prawdopodobieństwami i nagrodami)
- Możemy wszakże wykonywać pewne eksperymenty w naszym systemie, w wyniku których zdobywamy wiedzę **jak nam poszło**

## Uwaga

Zauważmy, że to pasuje do naszych samochodzików z rozmytymi stanami (eksperiment przeprowadzamy na prawdziwym modelu, ale obserwujemy model „rozmyty”)

# Ogólny schemat uczenia ze wzmacnieniem

Dla  $t \in 1, 2, 3, \dots$

- Wybieramy akcje  $a_t = \pi_{act}(s_{t-1})$  (**jak?**)
- Wykonujemy akcję i obserwujemy nowy stan  $s_t$
- Uaktualniamy parametry (**jak?**)

- Estymujemy model podczas eksperymentów
- Rozwiążujemy *wyszacowane* MDP.

## Szacowany MDP

- $\hat{T}(s, a, s') = \frac{\text{cnt}(s, a, s')}{\text{cnt}(s, a)}$
- Nagroda: średnie  $r$  dla zaobserwowanych  $s$  a  $r$   $s'$

- Jaką polityką mamy badać świat?
  - a) Wybierającą akcje losowo (strata czasu?)
  - b) Wybierającą akcje prowadzącą do stanu o najlepszej wartości  $V$  (ale możemy się zafiksować na nieoptymalnej ścieżce)
- Wybór między a) i b) to wybór między eksploracją i eksploatacją
  - Pamiętamy: strategia  $\varepsilon$ -zachłanna.

Możemy przeplatać etapy wyznaczania modelu i rozwiązywania MDP (bo w kolejnych iteracjach mamy możliwość wykorzystania lepszej strategii eksploatacyjnej).

# Pytanie kontrolne

Dlaczego przeplatanie estymacji modelu i wyznaczania optymalnej polityki może nam pomóc osiągnąć lepszy rezultat?

Mówiliśmy o metodach Monte Carlo, w których przeprowadzamy eksperymenty (losowe przebiegi), żeby estymować (nieznane) parametry MDP.

- Nowy cel: od razu liczyć  $Q(s, a)$ , nie przejmując się tworzeniem modelu.
- Zaczniemy od obliczenia  $Q_\pi(s, a)$

## Definicja (przypomnienie)

$Q_\pi(s, a)$  to oczekiwana sumaryczna nagroda, jaką otrzymamy wykonując w stanie  $s$  akcję  $a$ , a następnie postępując zgodnie z polityką  $\pi$

- Użyteczność (dla konkretnego przebiegu):

$$u_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots$$

- $\hat{Q}_\pi(s, a) = \text{średnie } u_t, \text{ gdzie } s_{t-1} = s, a_t = a$

- Zamiast liczyć średnią z całości, można myśleć o aktualnianiu średniej wraz z pojawieniem się kolejnej informacji.
- Niech:  $\eta = \frac{1}{1 + \text{cnt}(s, a)}$
- $\hat{Q}_\pi(s, a) \leftarrow (1 - \eta)\hat{Q}_\pi(s, a) + \eta u$  (gdzie  $u$  jest użytecznością zaobserwowaną w konkretnym przebiegu)

Sprawdźmy, czy to się zgadza.

$$\frac{\text{cnt}(s, a)\hat{Q}_\pi(s, a)}{1 + \text{cnt}(s, a)} + \frac{u}{1 + \text{cnt}(s, a)}$$

$$\hat{Q}_\pi(s, a) \leftarrow (1 - \eta)\hat{Q}_\pi(s, a) + \eta u$$

- $u$  jest zaobserwowaną użytecznością
- $\hat{Q}_\pi(s, a)$  jest naszą predykcją.

Reguła ta minimalizuje odległość między predykcją a obserwacją.

## Uwaga

W informatyce często, rozwiązujeając jakieś zadanie, korzystamy z niedoskonałego (tymczasowego) rozwiązania, żeby rozwiązać zadanie lepiej.

## Przykład

Szukanie dobrych i złych słów (analizujemy wpisy na jakimś forum), na początku znamy kilka przykładowych dobrych i złych słów.

Będziemy używać  $Q$  (poprzedniej wartości) do obliczenia nowego  $Q$

# Bootstraping: SARSA

Obserwujemy ciąg akcji i nagród:

$$s_0, a_1, r_1, s_1, a_2, r_2, s_2, \dots$$

- Uaktualnianie Monte Carlo:

$$\hat{Q}_\pi(s, a) \leftarrow (1 - \eta)\hat{Q}_\pi(s, a) + \eta u$$

- SARSA (obserwujemy  $s, a, r, s', a'$ ):

$$\hat{Q}_\pi(s, a) \leftarrow (1 - \eta)\hat{Q}_\pi(s, a) + \eta(r + \gamma \hat{Q}_\pi(s', a'))$$

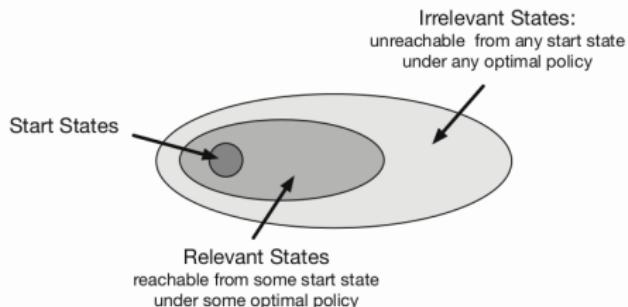
W algorytmie SARSA zamiast konkretnego (zaobserwowanego) **u** bierzemy zaobserwowaną jego **i** pierwszą część (**r**) i estymowaną resztę (zielony jest **cel**)

## Uwaga

Nie musimy czekać do końca epizodu, żeby uaktualnić wartość  $Q$ !



# Value iteration vs. SARSA



źródło: Sutton, Reinforcement Learning. An introduction

- VI liczy wartości dla stanów „nieoptymalnych”
- VI liczy wartości dla stanów nieosiągalnych (łatwo wymyślić dla autek taką kombinację prędkości i położenia, która jest bezużyteczna)

W momencie, gdy operujemy przebiegami, być może sensownymi, to koncentrujemy się na estymacji rzeczy użytecznych (a na pewno na **osiągalnych!**)

## Uwaga

SARSA estymuje  $Q_\pi(s, a)$ . Najbardziej naturalnym celem jest znajomość  $Q_{\text{opt}}$ .

- Algorytm umożliwiający bezpośrednie obliczanie  $Q_{\text{opt}}$  to właśnie **Q-learning**.
- Również radzimy sobie bez modelu.

# Q-learning

Standardowy kształt reguły:

$$Q(s, a)_{\text{opt}} \leftarrow (1 - \eta)Q(s, a)_{\text{opt}} + \eta \text{ cel}$$

Celem jest  $r + \gamma V_{\text{opt}}(s')$

Natomiast:

$$V_{\text{opt}}(s') = \max_{a' \in \text{Actions}(s')} Q_{\text{opt}}(s', a')$$

## Algorytm Q-learning

Dla zaobserwowanych  $s, a, r, s'$  (dla czytelności bez opt):

$$Q(s, a) \leftarrow (1 - \eta)Q(s, a) + \eta(r + \gamma \max_{a' \in \text{Actions}(s')} Q(s', a'))$$

- Jeżeli chcemy zachowywać się optymalnie powinniśmy wiedzieć coś o każdej parze  $(s, a)$
- Istnieją dwie możliwości:
  1. Rzeczywiście mamy szansę (w granicy) wygenerować przebieg z każdą parą  $(s, a)$
  2. Umiemy jakoś generalizować i wywnioskować coś na temat  $(s, a)$  korzystając z **podobnego**  $(s', a')$

Koniec części II

Obok wnioskowania i przeszukiwania jeden z głównych silników sztucznej inteligencji.

Użyteczne między innymi w sytuacjach o których ostatnio mówiliśmy, gdy nie chcemy pamiętać wartości  $Q(s, a)$  lecz umieć ją obliczyć

Fragment oceny opisowej ze świadectwa dziecka  
... umie odróżniać psa od kota.

Dane uczące



# Uczenie z nadzorem

Fragment oceny opisowej ze świadectwa dziecka  
... umie odróżniać psa od kota.

Dane uczące i dane testowe



- Spróbujemy usystematyzować nasze intuicje związane z uczeniem.
- Co wiemy:
  - a. Mamy przykłady, próbujemy je uogólnić.
  - b. Jeden z podstawowych zadań: klasyfikacja, czyli przypisanie przypadkowi jego klasy.
  - c. Przykłady:
    - Ocena, czy mail należy do spam czy też nie-spam.
    - Wybór rasy dla zdjęcia psa
    - Czy napis jest adresem e-mail, url-em, nazwą firmy, imieniem i nazwiskiem, czymś innym?

- Oczekiwanym wynikiem może być liczba rzeczywista.
- Przykłady:
  - predycja ceny nieruchomości,
  - ocena masy ciała (gdy znamy płeć i wzrost),
  - przewidywanie zużycia wody (dla MPWiK), gdy znamy temperaturę i dzień tygodnia

# Cechy kota i psa

## Uwaga

Zadanie rozróżnienia kota i psa byłoby łatwiejsze, gdybyśmy mieli dane nie obrazki, lecz cechy zwierzęcia

## Przykłady?

- masa ciała,
- odległość między oczami,
- odległość nosa i oka,
- długość włosów,
- długość wąsów,
- długość ogona

użyteczne mogłyby być też na przykład proporcje różnych cech i inne **wtórne cechy** (wyliczone z podstawowych)

# Cechy (wektor cech)

- Abstrakcyjny **obiekt** możemy zamienić na **wektor cech**.
- Dla (zabawkowego) klasyfikatora **czy-email?**, możemy mieć:
  - Czy długość większa od 10?
  - Jaki procent znaków to znaki alfanumeryczne?
  - Czy zawiera @?
  - Czy kończy się na .com (i tak dalej)

Cechami mogą być też na przykład wartości składowych pikseli, kolejne wartości pliku wave, zbiory pomiarów wszystkich wodomierzy z ostatniej doby, itd.

Dla obiektu **x** wektor cech oznaczamy często jako  
 $(\phi_1(x), \dots, \phi_n(x))$ .

- Dane uczące – zbiór przykładów, często w postaci:

(wektor-cech1, wynik1)

(wektor-cech2, wynik2)

(wektor-cech3, wynik3)

...

# Podział dostępnych danych

## Definicja 1

Zbiór **uczący** jest podstawowym zbiorem, który będziemy wykorzystywać do zdobywania wiedzy o problemie.

## Definicja 2

Zbioru **walidacyjnego** używamy do wyboru rodzaju algorytmu lub do wyboru hiperparametrów algorytmu.

## Definicja 3

Zbiór **testowy** używany jest **tylko** do ostatecznego testu, który ma nas przekonać, jak dobrze uogólnia rzeczywistość nasz mechanizm.

Do zbioru testowego nawet nie загlądamy, nie analizujemy błędów, itd!

Czas na przerwę

# Klasyczne zadanie klasyfikacji obrazów

**MNIST** jest zbiorem około 60K czarnobiałych obrazków  $28 \times 28$  zawierających ręcznie pisane cyfry.

Jest on powszechnie używany do testowania różnych algorytmów uczenia (głównie klasyfikacji, ale nie tylko)

# MNIST – przedstawienie danych

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1  
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2  
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3  
4 4 4 4 4 4 4 4 4 4 4 4 4 4 4  
5 5 5 5 5 5 5 5 5 5 5 5 5 5 5  
6 6 6 6 6 6 6 6 6 6 6 6 6 6 6  
7 7 7 7 7 7 7 7 7 7 7 7 7 7 7  
8 8 8 8 8 8 8 8 8 8 8 8 8 8 8  
9 9 9 9 9 9 9 9 9 9 9 9 9 9 9

- Naturalnym rozwiążaniem jest stworzenie **wzorca** (lub wzorców) dla każdej cyfry.
- Jak to zrobić?

## Dwa warianty

- ① Jeden wzorzec dla wszystkich obrazków danej cyfry.
- ② Wiele wzorców (nawet: **każda cyfra wzorcem**)

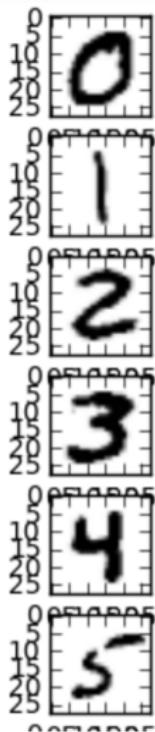
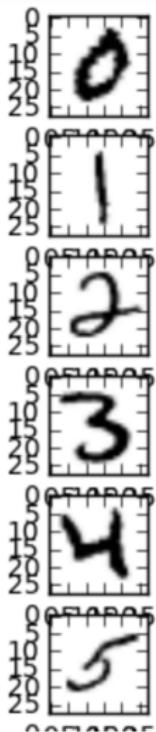
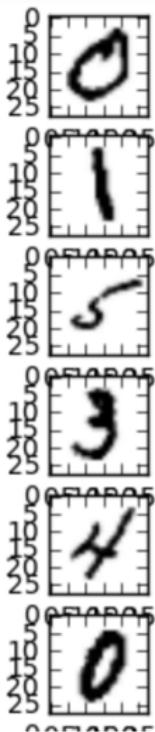
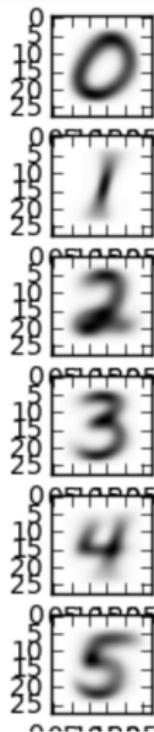
# Jeden wzorzec dla cyfry

- Naturalnym wzorcem może być średnia wszystkich egzemplarzy danej cyfry
- Przy klasyfikacji obrazka  $\mathbf{o}$  wybieramy wzorzec  $\mathbf{w}$  najbardziej podobny do  $\mathbf{o}$
- Co to znaczy podobny?
  - Wysoki iloczyn skalarny?
  - *Wysoki iloczyn skalarny znormalizowanych wektorów?*
  - Niewielka odległość euklidesowa (a może jakaś inna)?

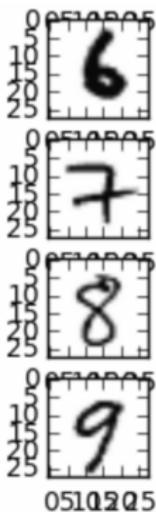
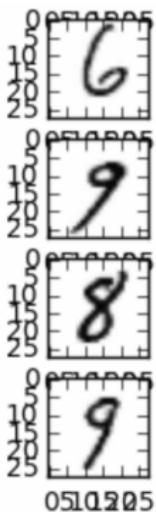
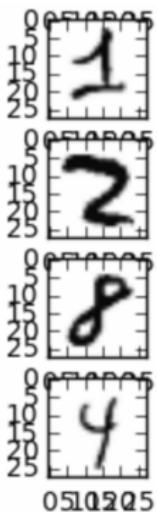
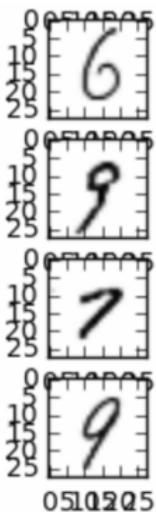
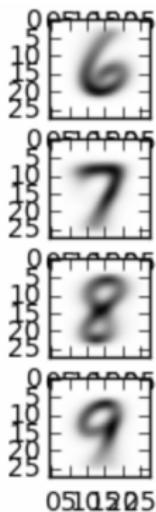
## Wyniki eksperymentu

Poprawność klasyfikacji to około **82.1%** (dla iloczynów skalarnych znormalizowanych wektorów)

# Wyniki klasyfikacji z „wzorcami średnimi”

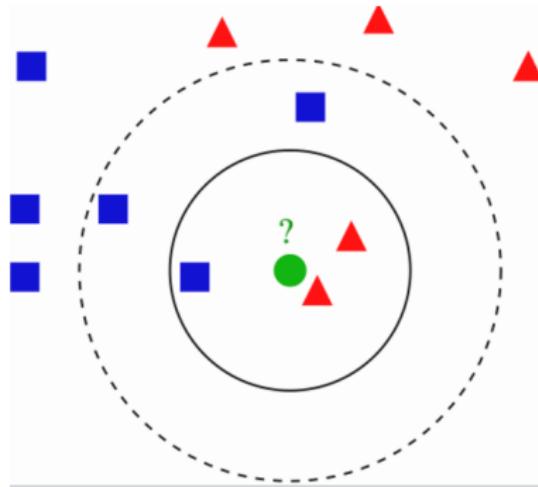


# Wyniki klasyfikacji z „wzorcami średnimi”



# K najbliższych sąsiadów. KNN

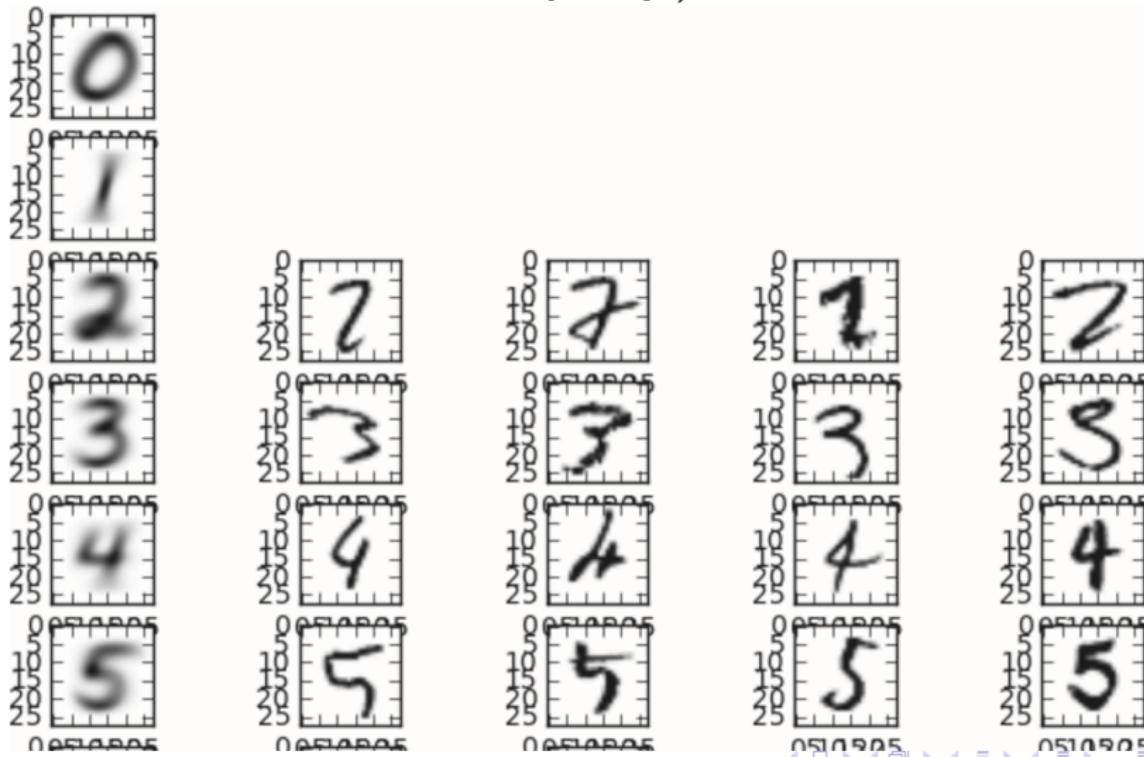
- Pamiętamy wszystkie wektory ze zbioru uczącego.
- Klasyfikując obrazek znajdujemy  $K$  najbliższych sąsiadów i pozwalamy im głosować



- Podobieństwo mierzymy iloczynem skalarnym znormalizowanych wektorów (czyli **cosinusem**)
- Testujemy na próbce (bo inaczej trwa bardzo długo)
- $K = 3$
- Wyniki:
  - Zbiór uczący: **98.55%**
  - Zbiór testowy: **około 97%**

# MNIST i KNN. Przykładowe błędy

Z tymi cyframi mieliśmy problemy (zaznaczona prawidłowa klasyfikacja)



# MNIST i KNN. Przykładowe błędy

Z tymi cyframi mieliśmy problemy (zaznaczona prawidłowa klasyfikacja)



# O uczeniu maszynowym i sieciach neuronowych

Paweł Rychlikowski

Instytut Informatyki UWr

7 maja 2021

# Uczenie maszynowe. Przypomnienie

- Rozważaliśmy uczenie **z nadzorem** (to znaczy sytuację, gdy mamy pewne przykłady i chcemy je uogólnić)
- Rozważamy dwie ważne klasy takich zadań:
  - ① Klasyfikacji (wybór klasy dla przykładu, zbiór klas jest skończony i niezbyt duży)
  - ② Regresji (wybór wartości liczbowej dla przykładu)

## Uwaga

### Dopasowanie wzorców (pattern matching)

Rozważaliśmy dwa rodzaje wzorców:

- Średnia cyfra (jeden wzorzec dla każdej cyfry)
- Każda cyfra jest wzorcem (do klasyfikacji cyfry  $D$  znajdujemy  $K$  najbliższych wzorców i przeprowadzamy głosowanie)

- Klasyfikacji (czy regresji) możemy dokonywać na bazie wartości:

$$\sum_{i=0}^N w_i \phi_i(x)$$

- Cechami (dla MNIST-a) są po prostu wartości kolejnych pikseli
- Jedna funkcja „wystarcza” dla binarnego zadania typu: **czy cyfra jest piątką? (i jak bardzo)**
- Do rozwiązywania MNIST-a potrzebujemy 10 takich funkcji, wybieramy cyfrę dla tej funkcji, która zwraca największą wartość.

- Możemy zdefiniować zadanie uczenia (regresji) dla Reversi:

*Widząc sytuację na planszy w ruchu 20 postaraj się przewidzieć zakończenie gry.*

- Cechy: binarne cechy mówiące o zajętości pola.

- Przykładowo:

Czy pole (4,5) jest czarne?

Czy pole (1,1) jest białe?

## Uwaga

Taki mechanizm byłby użyteczną funkcją heurystyczną, do użycia np. w algorytmie MiniMax.

## Definicja

Funkcja **kosztu** (loss) opisuje, jak bardzo **niezadowoleni** jesteśmy z działania naszego mechanizmu (klasyfikatora, przewidywacza wartości).

Funkcja kosztu jest określona na: danych uczących ( $x, y$ ) oraz wagach (parametrach klasyfikatora). Przy czym dane uczące traktujemy jako parametr, a wagi – jako właściwe argumenty.

Przykładowa funkcja kosztu: **błąd średniokwadratowy**

# Funkcja kosztu (2)

Średniokwadratowa funkcja straty

$$\text{TrainLoss}(\mathbf{w}) = \frac{1}{|D_{\text{train}}|} \sum_{(x,y) \in D_{\text{train}}} (f_{\mathbf{w}}(x) - y)^2$$

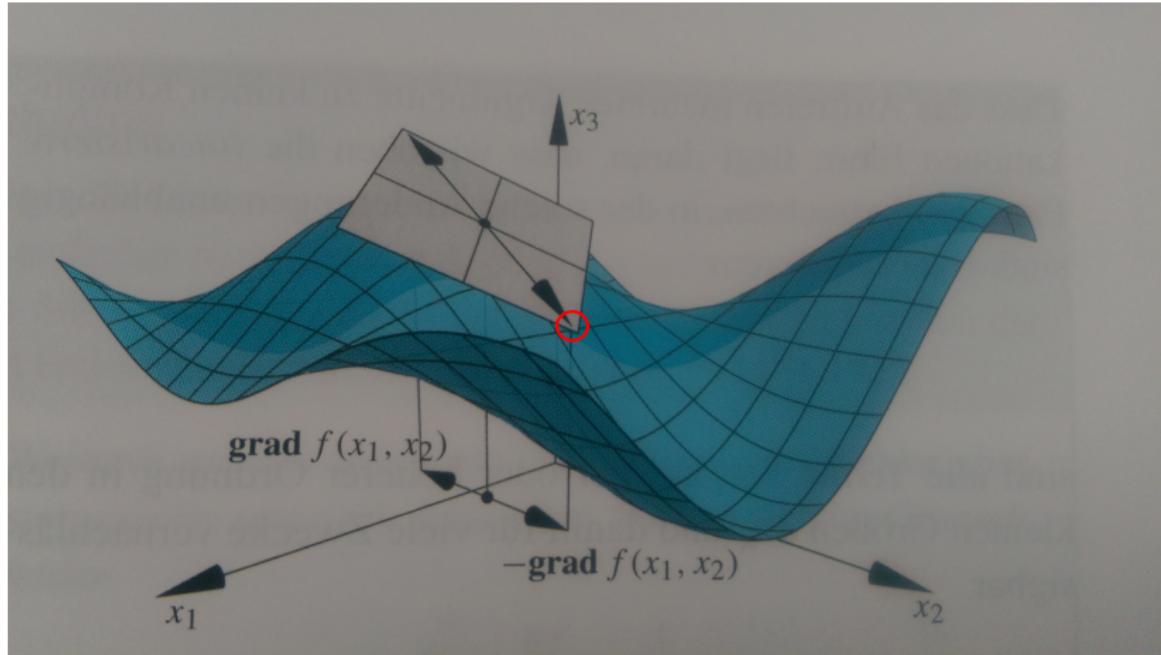
Wariant liniowy:

$$f_{\mathbf{w}}(x) = \sum_{i=0}^N w_i \phi_i(x)$$

- Gradient jest wektorem pochodnych cząstkowych.
- Wskazuje kierunek największego wzrostu funkcji.

$$\nabla f(p) = \begin{bmatrix} \frac{\partial f}{\partial x_1}(p) \\ \vdots \\ \frac{\partial f}{\partial x_n}(p) \end{bmatrix}$$

Wizualizacja gradientu w dla funkcji  $\mathcal{R}^2 \rightarrow \mathcal{R}$



## Wzór

Pochodna po  $w_i$  (dla liniowej funkcji  $f_w$ )

$$\frac{1}{|D_{\text{train}}|} \sum_{(x,y) \in D_{\text{train}}} 2 \cdot (f_w(x) - y) \cdot w_i \phi_i(x)$$

- Obliczenie gradientu wymaga przejścia przez cały zbiór uczący
- Maksymalizacja funkcji: dodawanie gradientu przemnożonego przez małą stałą (minimalizacja: odejmowanie)

## Stochastic Gradient Descent

Obliczamy nie cały gradient, tylko jego składnik, związany z jednym egzemplarzem danych uczących i jego dodajemy (przemnożonego przez stałą)

# Koniec części I

- Wybierzemy absolutne minimum tego, co należy wiedzieć o sieciach neuronowych
- Oczywiście nie będzie to w pełni kompletna wiedza.

- Neuron to funkcja  $f : \mathcal{R}^n \rightarrow \mathcal{R}$

$$f(x_1 \dots x_n) = \sigma\left(\sum_{i=1}^n w_i x_i + b\right)$$

- $\sigma$  jest jakąś ustaloną funkcją nieliniową, raczej rosnącą, raczej różniczkowalną, na przykład:  $\max(0, v)$ , albo  $\tanh(v)$
- Wygodna (jak za chwilę zobaczymy) jest notacja wektorowo-macierzowa, w niej mamy:

$$f(\mathbf{x}) = \sigma(\mathbf{w}^T \cdot \mathbf{x} + b)$$

- Warstwa to funkcja  $\mathcal{R}^n \rightarrow \mathcal{R}^m$ .
- Najbardziej typowa warstwa wyraża się wzorem:

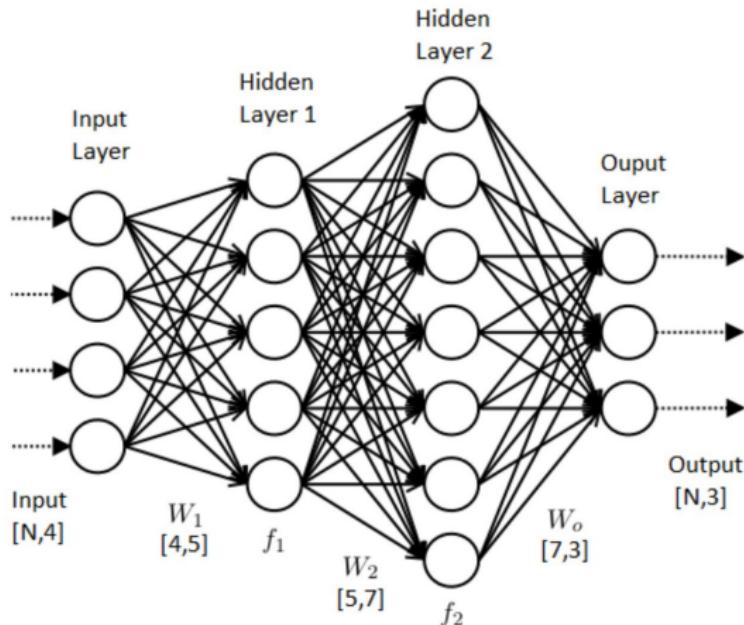
$$L(\mathbf{x}) = \sigma(\mathbf{Wx} + \mathbf{b})$$

- **Uwaga:**  $\mathbf{W}$  jest macierzą wag (złożoną z wektorów wag), a  $\sigma(y_1 \dots y_m) = (\sigma(y_1) \dots \sigma(y_m))$

## Definicja

Sieć neuronowa typu **MLP** jest złożeniem warstw (z różnymi macierzami wag dla każdej warstwy).

# Slajd 2b. Prosta sieć neuronowa



Źródło: VIASAT (<https://medium.com/coinmonks/the-artificial-neural-networks-handbook-part-1-f9ceb0e376b4>)

## Zadanie

Danymi jest ciąg  $(\mathbf{x}_i, \mathbf{y}_i)$  opisujący porządkowane zachowanie sieci  $S$  oraz architektura tejże sieci (liczba warstw, ich wymiary, funkcja/funkcje  $\sigma$ ).

Chcemy tak dobrać parametry ( $\mathbf{W}_k$  oraz  $\mathbf{b}_k$ ) żeby dla każdego  $i$

$$S(\mathbf{x}_i) \approx \mathbf{y}_i$$

## Funkcja kosztu

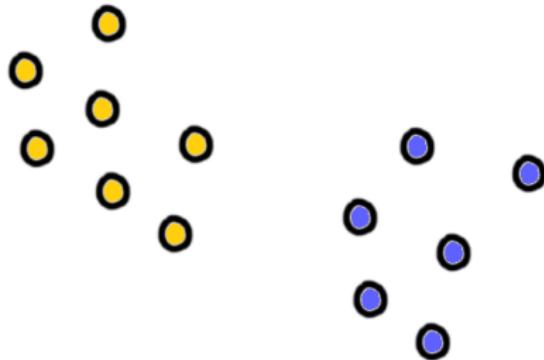
Powysze zadanie formalizujemy jako zadanie znalezienia takich parametrów, że **koszt** błędów jest jak najmniejszy. Przykładowo, jeżeli wyjściem jest liczba, to możemy wybrać:

$$\text{Loss}(\theta) = \sum_i^n (S_\theta(\mathbf{x}_i) - y_i)^2$$

## Ogólne założenia (przypadek dwóch klas)

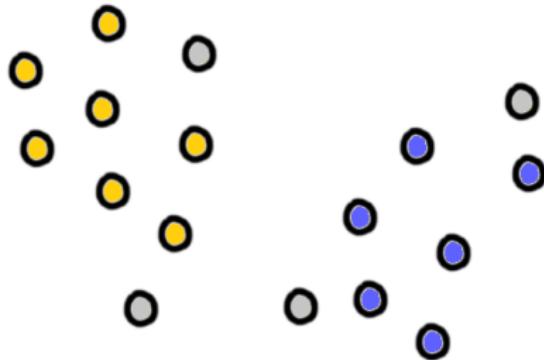
Mamy jakiś zbiór przykładów **pozytywnych** i **negatywnych**, interesuje nas mechanizm, który będzie poprawnie klasyfikował nieznane przykłady.

# Klasyfikacja w $\mathcal{R}^2$



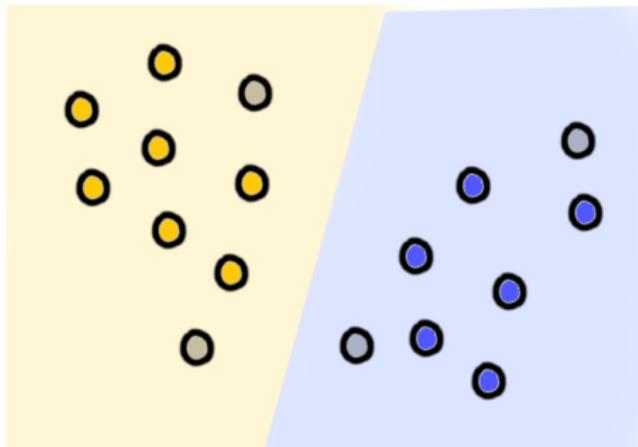
- Dane punkty wraz z informacją o kolorze.
- Mechanizm powinien umieć określić kolor nieznanych punktów.
- Możemy o tym myśleć, jako o „kolorowaniu płaszczyzny”

# Klasyfikacja w $\mathcal{R}^2$



- Dane punkty wraz z informacją o kolorze.
- Mechanizm powinien umieć określić kolor nieznanych punktów.
- Możemy o tym myśleć, jako o „kolorowaniu płaszczyzny”

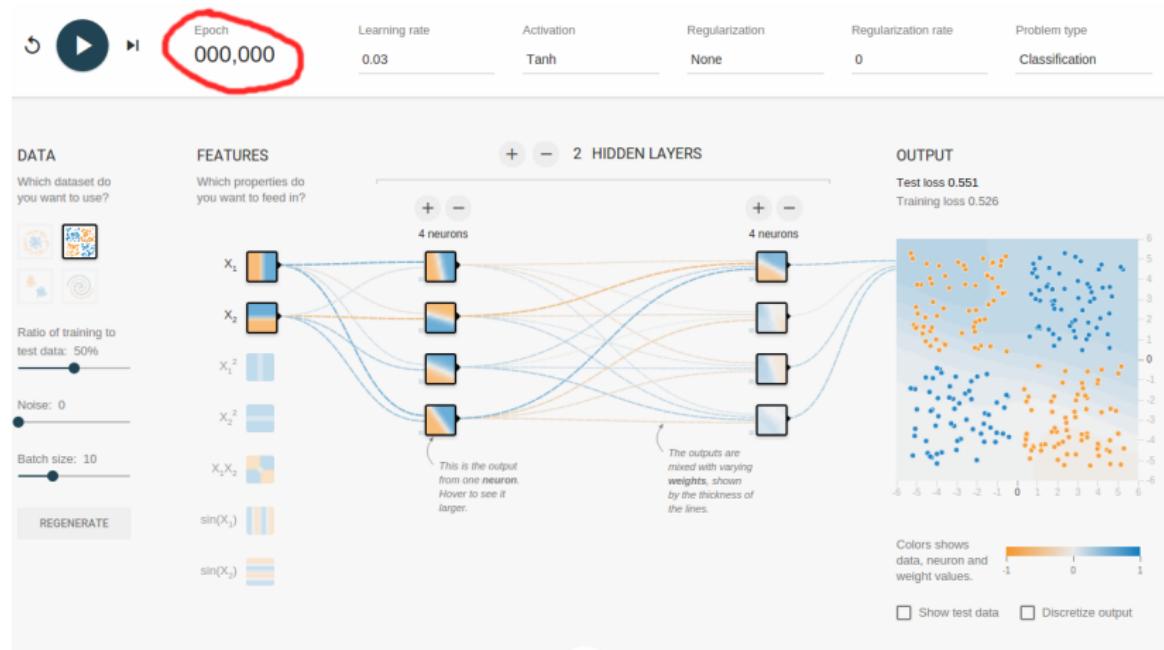
# Klasyfikacja w $\mathcal{R}^2$



- Dane punkty wraz z informacją o kolorze.
- Mechanizm powinien umieć określić kolor nieznanych punktów.
- Możemy o tym myśleć, jako o „kolorowaniu płaszczyzny”

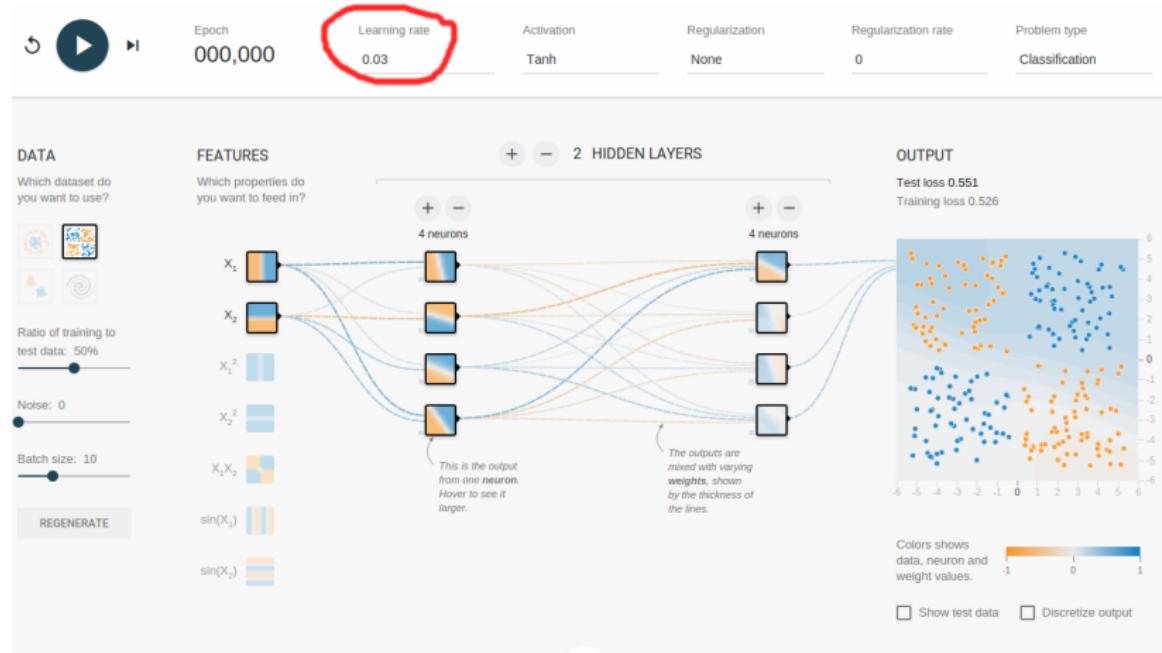
Spróbujmy poeksperymentować chwilę z Tensorflow Playground

# Plac zabaw dla tensorflow. Ważne pojęcia (1)



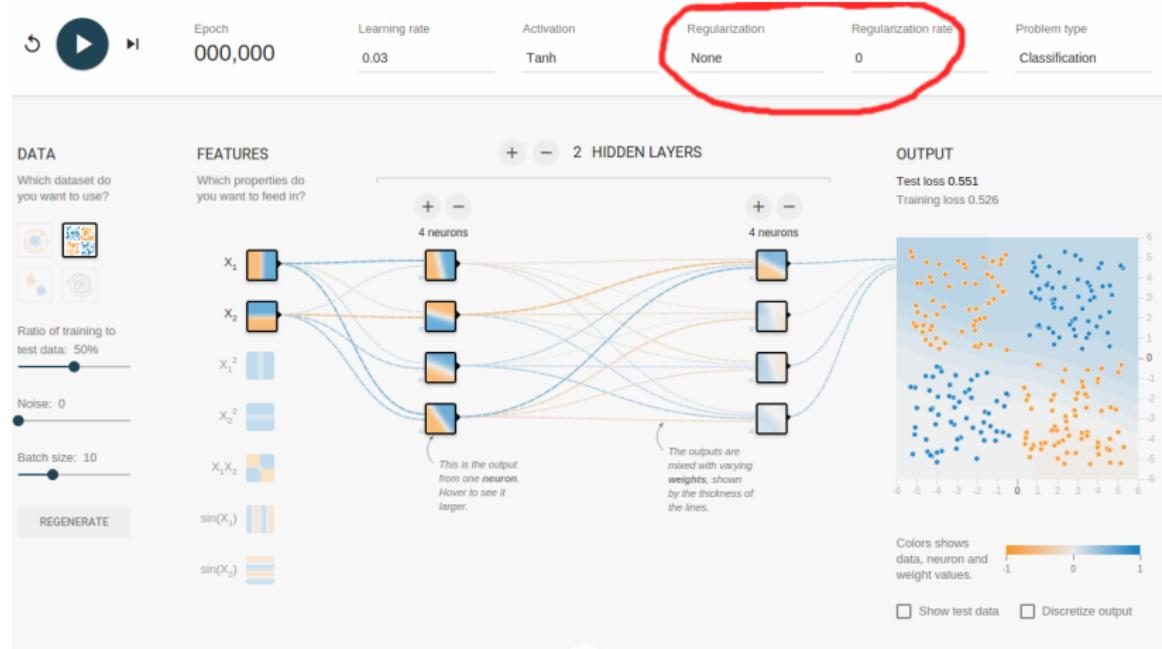
**Epoka:** etap uczenia, w którym uwzględnione są wszystkie dane uczące.

# Plac zabaw dla tensorflow. Ważne pojęcia (2)



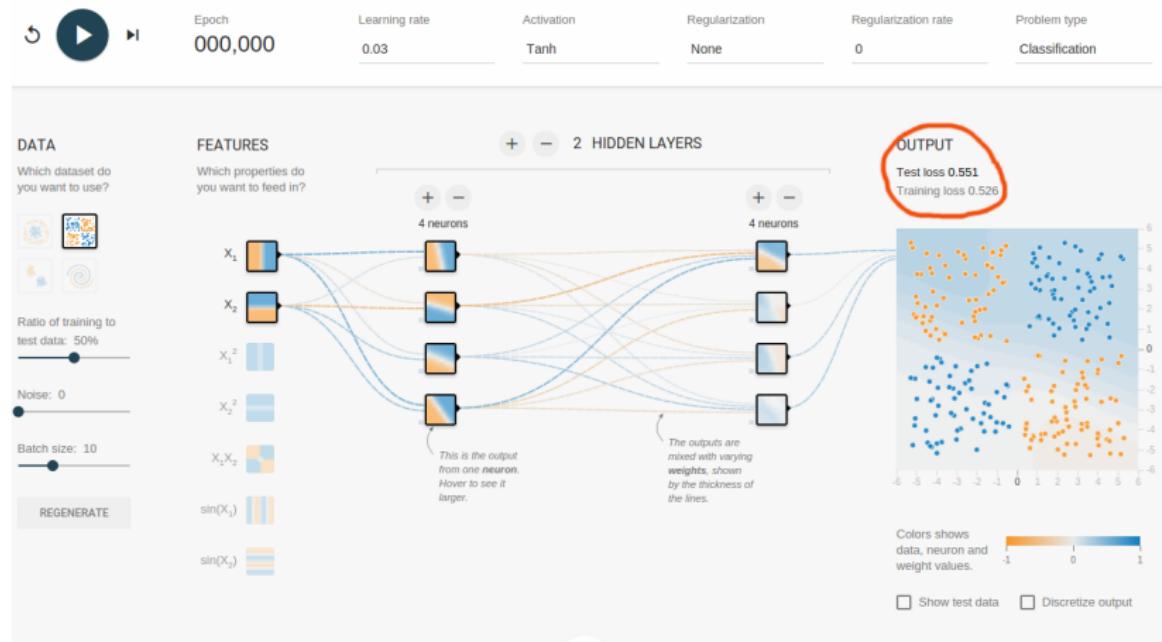
**Learning rate:** stała przez którą mnożone są delty wag. Za duża może dać chaotyczne zachowanie, za mała: bardzo wolny postęp.

# Plac zabaw dla tensorflow. Ważne pojęcia (3)



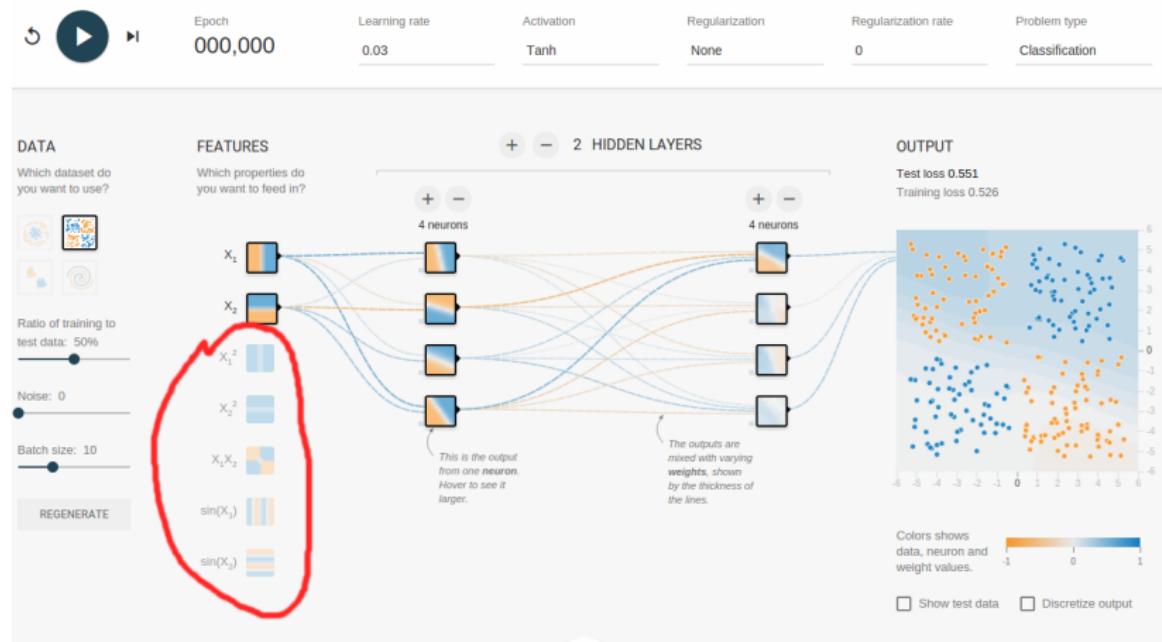
**Regularyzacja:** dołożenie do uczenia wymagania, by wagi nie były zbyt duże. Może dać większą stabilność uczenia (zob. tablica).

# Plac zabaw dla tensorflow. Ważne pojęcia (4)



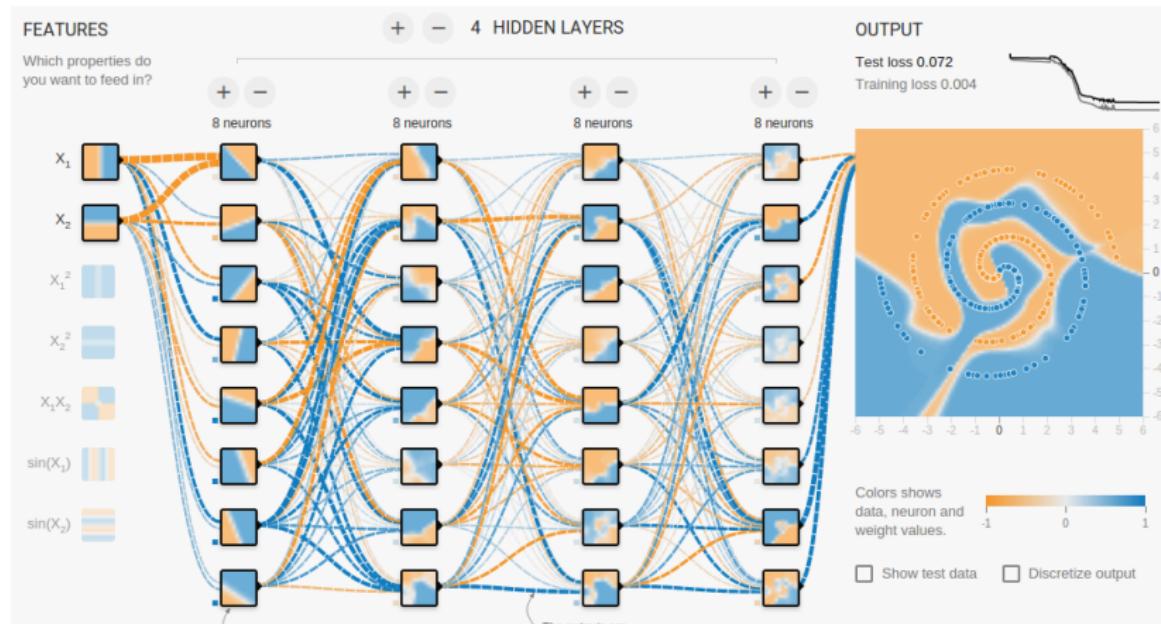
**Test loss/training loss:** wartość kosztu dla zbioru testowego i uczącego (oczywiście pierwsza zawsze większa).

# Plac zabaw dla tensorflow. Ważne pojęcia (5)



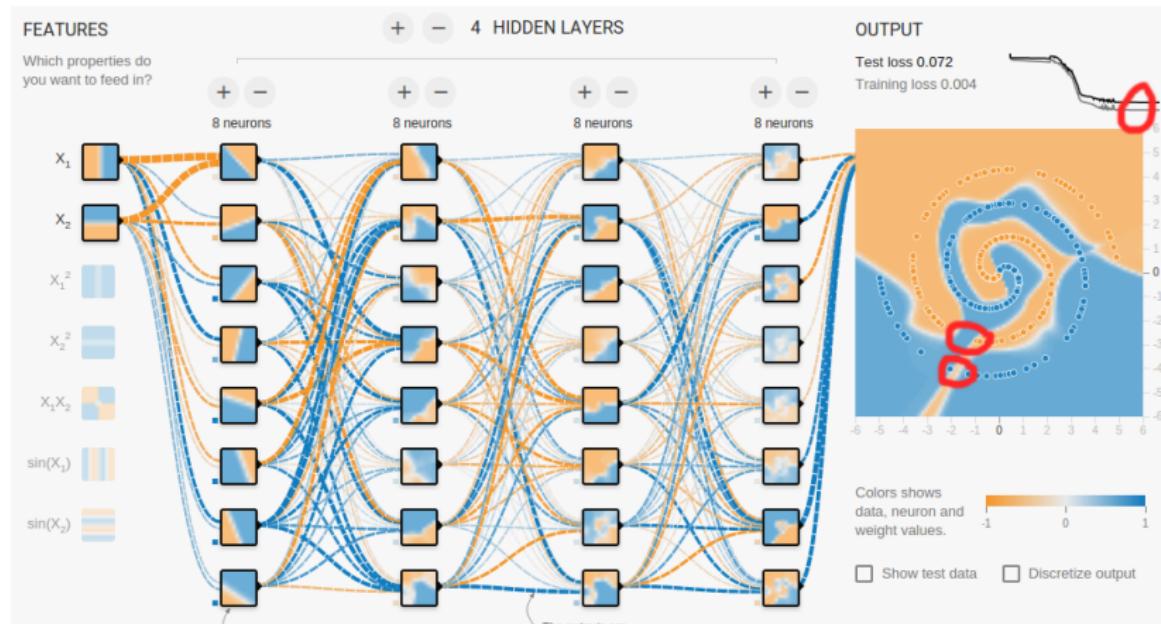
**Feature engineering:** proces tworzenia własnych cech dla konkretnych przypadków. Dobre cechy mają [związek z zadaniem](#).

# Plac zabaw dla tensorflow. Ważne pojęcia (6)



**Przeuczenie (overfitting):** sytuacja, w której sieć dostosowuje się do **nieistotnych** fluktuacji danych uczących, co pogarsza generalizację.

# Plac zabaw dla tensorflow. Ważne pojęcia (6)



**Przeuczenie (overfitting):** sytuacja, w której sieć dostosowuje się do **nieistotnych** fluktuacji danych uczących, co pogarsza generalizację.

# Kodowanie wejścia

- Wejściem do sieci jest **wektor** (czyli ciąg liczb o ustalonej długości)
- W tym wektorze możemy zakodować wszystko:
  - obrazki (jak?)
  - teksty o ustalonej długości (jak?)
  - sytuację na planszy w Reversi (jak?)

## Kodowanie **one-hot**

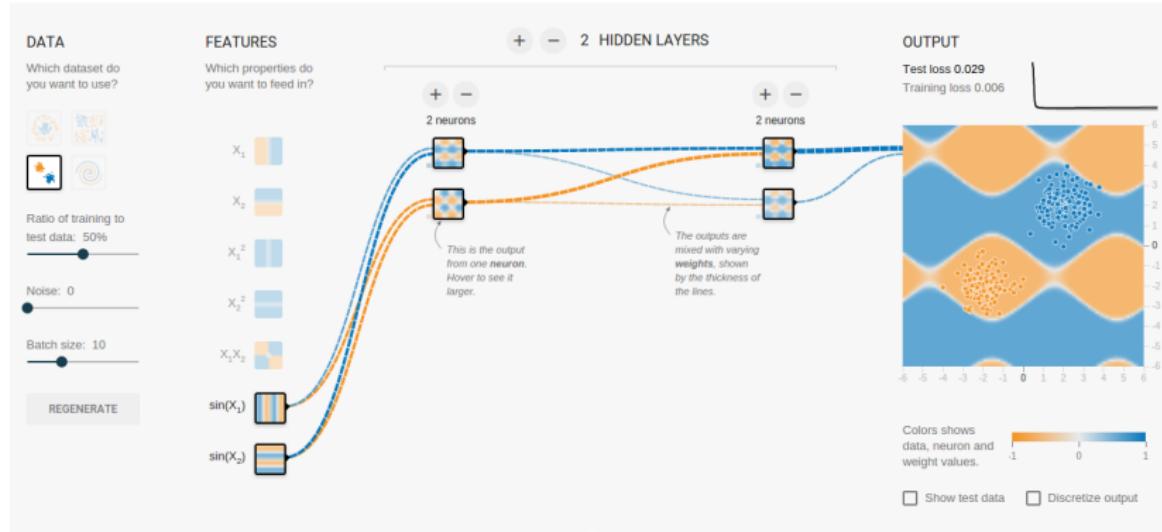
Sieci neuronowe lubią *rozwiąkłe* kodowanie, w którym liczbę  $i \in \{0, \dots, N - 1\}$  kodujemy jako  $(0, 0, 0, \dots, 1, \dots, 0, 0)$  (jedynka na  $i$ -tej pozycji).

- Zastanówmy się nad możliwymi kodowaniami obrazków, tekstów, fragmentów nagrań dźwiękowych, oraz planszy w reversi.
- Pamiętajmy, że możemy dowolnie tworzyć cechy dla przypadków testowych:
  - Kwantyzacja dla obrazów
  - Analiza Fouriera dla dźwięków
  - Tworzenie *pseudosłów* (rzeczownik, a-cja, ...)
  - ...

## Uwaga

Dodając cechy możemy przyśpieszyć uczenie, ale możemy też **zasugerować** sieci naszą wizję świata. Np. cecha w Reversi: **wynik jakiejś funkcji heurystycznej**.

# Sugerowanie cykliczności



Sieć w miarę poprawnie sklasyfikowała zbiór uczący, dobrze też go uogólnia, ale jest przekonana, że świat jest mozaiką. Nikt z nas, widząc te dane nie wyrobił sobie tego poglądu.

- Często chcemy, żeby sieć decydowała o jednej z  $K$  opcji (zadanie klasyfikacji).
- Rozmywamy ten wybór, prosząc o podanie rozkładu prawdopodobieństwa dla wszystkich  $K$  opcji.
- To tzw. **Softmax layer**, która przypisuje prawdopodobieństwo zależne od wielkości pobudzenia.

Wzór:

$$\text{Softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^d e^{z_j}}$$

Popatrzmy na to, jak działa funkcja Softmax.

```
>>> softmax([1,2,3])
```

```
[0.09003, 0.24472, 0.66524]
```

```
>>> softmax([1,2,3,10])
```

```
[0.00012, 0.00033, 0.00091, 0.99863]
```

```
>>> softmax([3,4,4])
```

```
[0.15536, 0.42231, 0.42231]
```

# Super łatwe sieci neuronowe

- Można wykorzystać bibliotekę **sklearn** (lub analogiczną), która implementuje **MLP** (czyli wielowarstwowy perceptron)
- Sieć definiujemy jednym konstruktorem z dużą liczbą parametrów (ale ufamy, że wartości domyślne są ok)

# Super łatwe sieci neuronowe

## Przygotowanie danych

```
from sklearn.neural_network import MLPClassifier
import random, pickle

# data: list of pairs (X,y)
# X: vector of floats/ints
# y in [v1, ..., vk]

random.shuffle(data)
N = len(data) / 6
test_data = data[:N]
dev_data = data[N:]

X = [x for (x,y) in dev_data]
y = [y for (x,y) in dev_data]
X_test = [x for (x,y) in test_data]
y_test = [y for (x,y) in test_data]
```



# Super łatwe sieci neuronowe (2)

## Uczenie sieci

```
# creating model
nn = MLPClassifier(hidden_layer_sizes=(60,60,10))

# training model
nn.fit(X,y)

print ('Dev-score', nn.score(X,y))
print ('Test-score', nn.score(X_test, y_test))

# writing model
with open('nn_weights.dat', 'w') as f:
    pickle.dump(nn, f)
```

# Super łatwe sieci neuronowe (3)

## Korzystanie z sieci

```
from sklearn.neural_network import MLPClassifier
import pickle

with open('nn_weights.dat') as f:
    nn = pickle.load(open(f))

x = data_vector

probabilities = nn.predict_proba([x])

prob0 = ys[0][0]
prob1 = ys[0][1]
```

# Podsumowanie

## Cons

- Oczywiście daje dużo mniejszą swobodę niż bardziej specjalizowane biblioteki.
- Nadaje się do tworzenia niezbyt dużych sieci
- Nie ma sieci splotowych, sieci rekurencyjnych, ...

## Pros

- Bardzo prosta w użyciu i wystarczająco szybka
- Ten sam (prawie) interfejs dla różnych mechanizmów:
  - `from sklearn.neighbors import KNeighborsClassifier as Classifier`
  - `from sklearn.tree import DecisionTreeClassifier as Classifier`
  - `from sklearn.svm import SVC as Classifier`
  - ... (i jeszcze kilkanaście innych)

# Uczenie maszynowe, różne warianty

Paweł Rychlikowski

Instytut Informatyki UWr

20 maja 2021

# Inne rodzaje uczenia

## Uwaga

Rozważaliśmy uczenie z nadzorem, czyli taki wariant, w którym dysponujemy dodatkowymi danymi (dotyczącymi np. prawidłowej klasyfikacji każdej próbki).

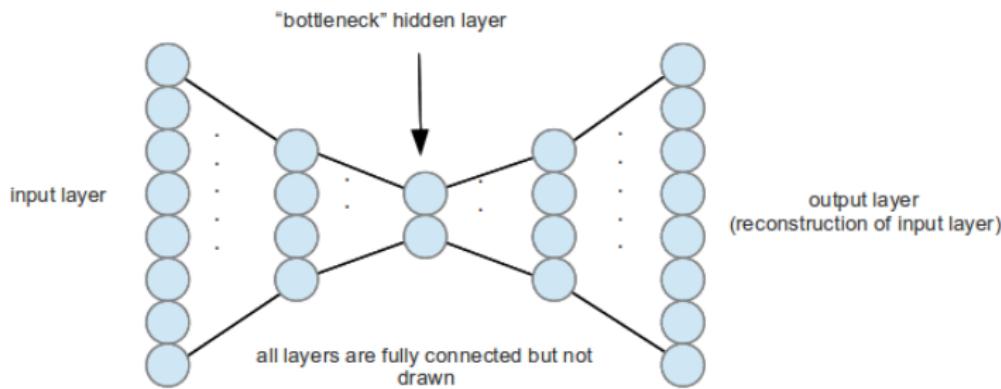
A co można zrobić, jeżeli mamy same próbki?

1. Nauczyć się generować podobne próbki (**autoenkoder**).
2. Pogrupować próbki (**algorytmy klasteryzacji**)
3. Narysować próbki (**algorytmy wizualizacji**)
4. Znaleźć **dziwne** próbki (**algorytmy wykrywania nieprawidłowości**, czyli **anomaly detection**)

Z wizualizacją i autoenkoderami związana jest **redukcja wymiarowości**

# Autoenkodery

- Tworzymy zadanie uczenia się z nadzorem (funkcji identycznościowej)
- Wariant jednowarstwowej funkcji liniowej jest skrajnie nieciekawy (bo?)
- Wielowarstwowa sieć, która ma część redukującą wymiar (coraz mniejsze warstwy) i analogiczną grupę warstw zwiększającą wymiar
- Może być użyteczna, bo tworzy wewnętrzną reprezentację obrazu



# Bardziej skomplikowane autoenkodery (NVIDIA Celebrities)

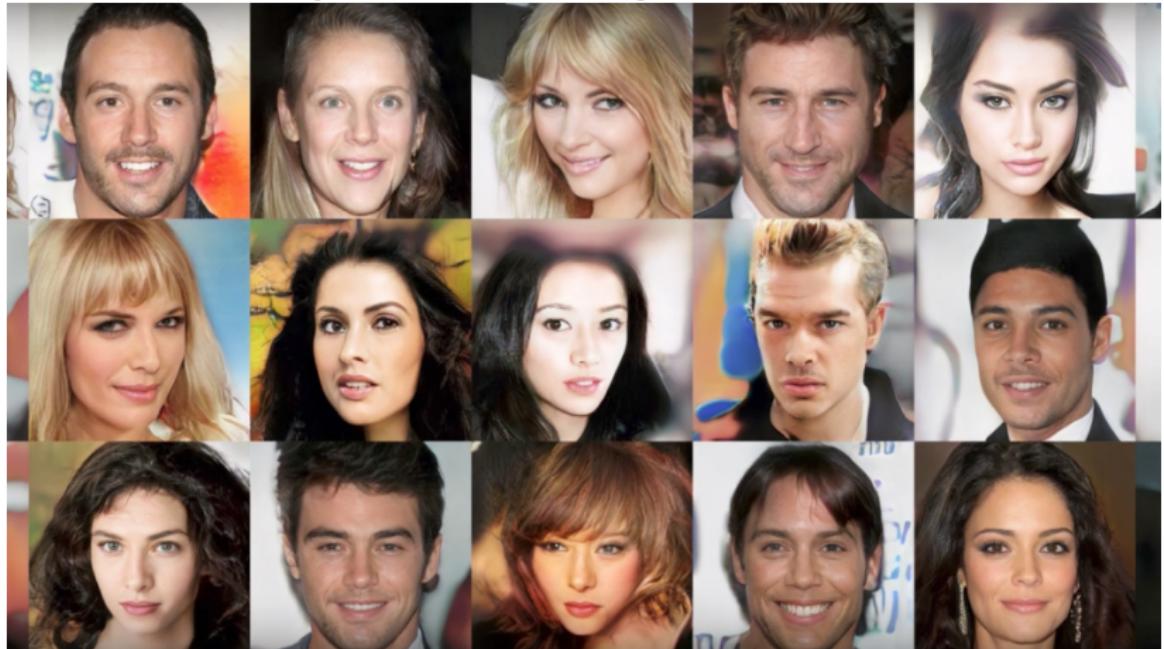
Ci ludzie nie dadzą Ci autografu (przykładowe twarze dla losowego zacisku)



źródło: <http://research.nvidia.com/>

# O twarzach (2)

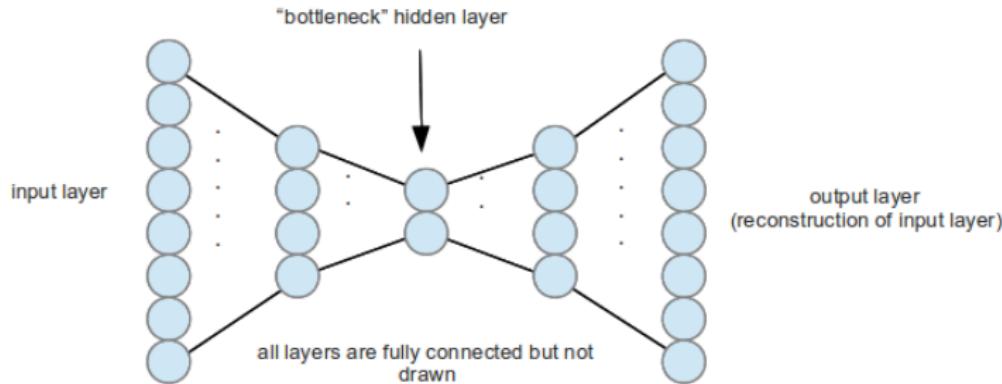
Oczywiście nie zawsze jest idealnie, bo:



źródło: <https://nerdist.com/nvidia-ai-headshots-fake-celebrities/>

# Anomalie (1)

Autoenkoder może być użyty do wykrywania anomalii.  
Autoenkodery radzą sobie dobrze z **typowymi próbками**.



**Nietypowe próbki będą miały duży błąd rekonstrukcji!**

## Definicja

**Klasteryzacja (grupowanie)** to zadanie identyfikacji w próbce uczącej naturalnych grup związanych ze sobą obiektów.

**Obiekt = wektor  $w \in \mathcal{R}^n$**

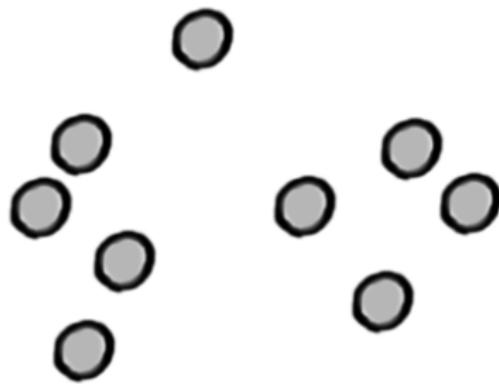
- Najprostszy wariant: chcemy otrzymać konkretną liczbę grup, powiedzmy  $K$
- Najprostszy algorytm: K-średnich (k-means)

Przez cały czas działania algorytmu pamiętamy  $K$  **prototypów** (czyli punktów będących reprezentantami grupy)

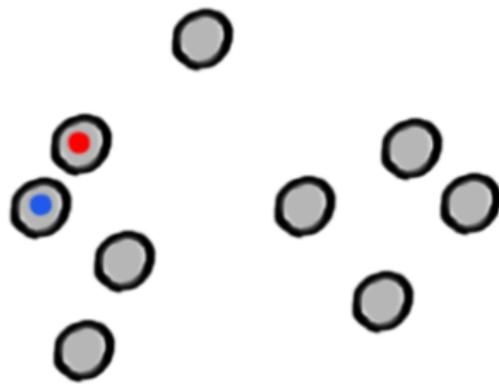
Algorytm przeplata dwie fazy:

- ① Przypisanie każdego punktu do najbliższego mu prototypu
- ② Obliczenie nowych prototypów jako **średnich** wszystkich punktów przypisanych do tego samego prototypu

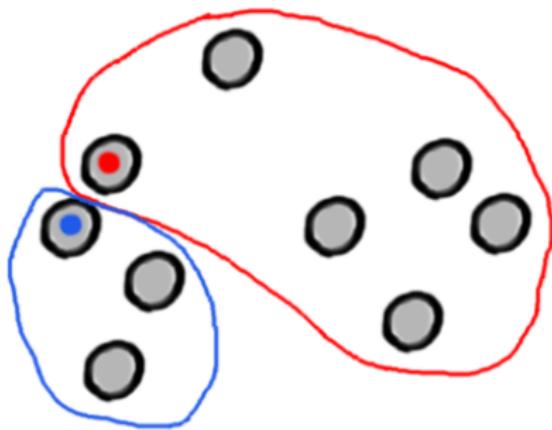
# Algorytm K-średnich. Przykład: K=2



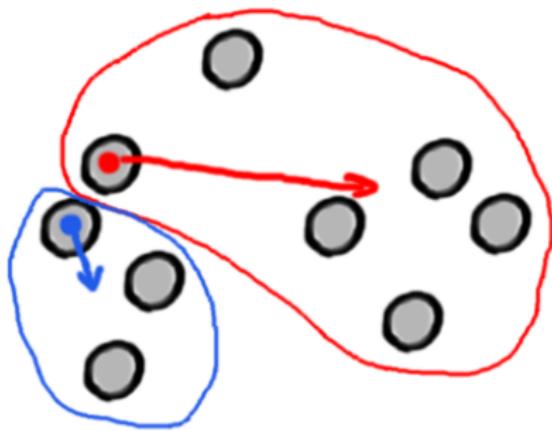
# Algorytm K-średnich. Przykład: K=2



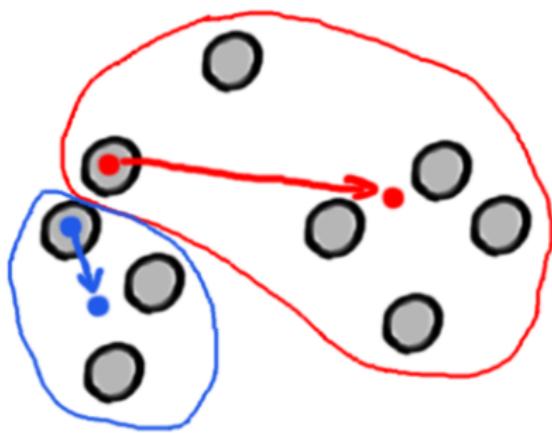
# Algorytm K-średnich. Przykład: K=2



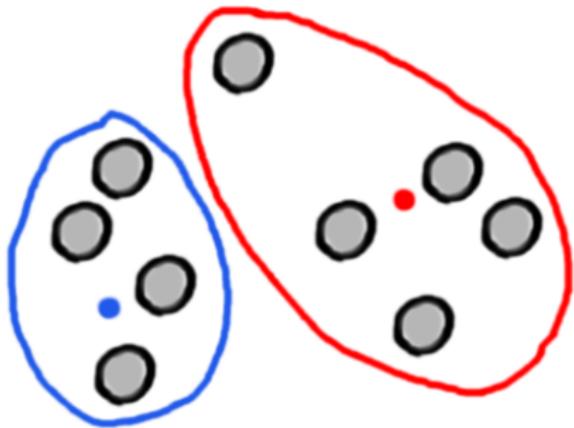
# Algorytm K-średnich. Przykład: K=2



# Algorytm K-średnich. Przykład: K=2



# Algorytm K-średnich. Przykład: K=2



Koniec części I

- Losujemy pewną liczbę punktów na płaszczyźnie, tak aby w naturalny sposób tworzyły klastry.
- Wybieramy początkowe centra z populacji punktów
- Obserwujemy, jak działa algorytm

Popatrzmy na demonstrację `kmeans.py`

Dodatkowe funkcje: `restart`, `new`

# Co oblicza algorytm $K$ -średnich?

## Cel

Chcemy, żeby **prototyp** przybliżał wszystkie przypisane mu elementy.

**Daleka analogia:** Prototyp jest takim **elektorem**, który przybliża poglądy swoich wyborców.

Każdy wyborca jest zadowolony, jeżeli ma elektora dobrze rozumiejącego jego preferencje.

# Co oblicza algorytm $K$ -średnich? (2)

- Interesuje nas, aby **każdy element, jak najmniej się różnił od swojego reprezentanta** (czyli średniej, prototypu, centroidu)
- Miara: błąd średniokwadratowy, czyli

$$\sum_{x \in \text{Dane}} (x - \text{reprezentant}(x))^2$$

## Definicja

Powyżej zdefiniowaną wielkość nazwiemy **błędem klasteryzacji**.

# $K$ -średnich jako minimalizacja błędu

## Etap przypisywania

Prototypy ustalone. Każdy egzemplarz trafia do bliższego prototypu (czyli błąd maleje).

## Etap liczenia średnich

Patrzymy na 1 klaster. Policzmy pochodną po  $c$  dla

$$\frac{1}{N} \sum_{i=1}^N (x_i - c)^2$$

Wychodzi:

$$\frac{2}{N} \sum_{i=1}^N (c - x_i)$$

Błąd klasteryzacji dla jednego klastra osiąga minimum w punkcie będącym średnią punktów klastra.



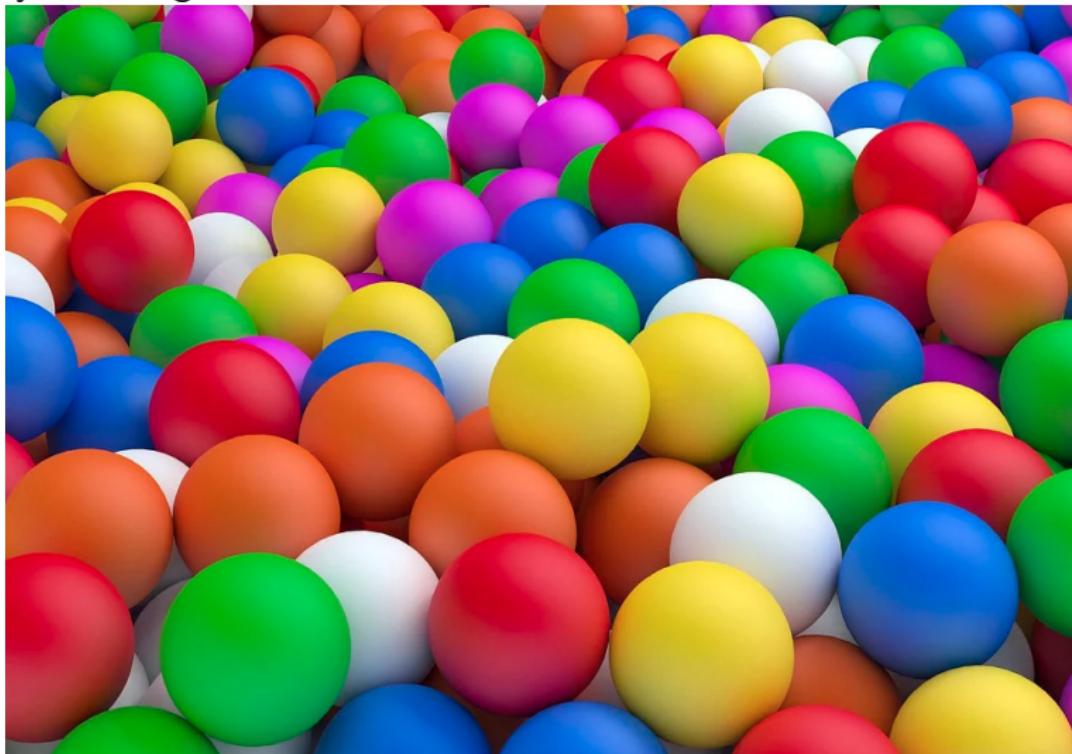
## Wniosek

Oba etapy nie zwiększają błędu (tzn. jeżeli coś robią, to błąd maleje). Czyli osiągamy lokalne minimum.

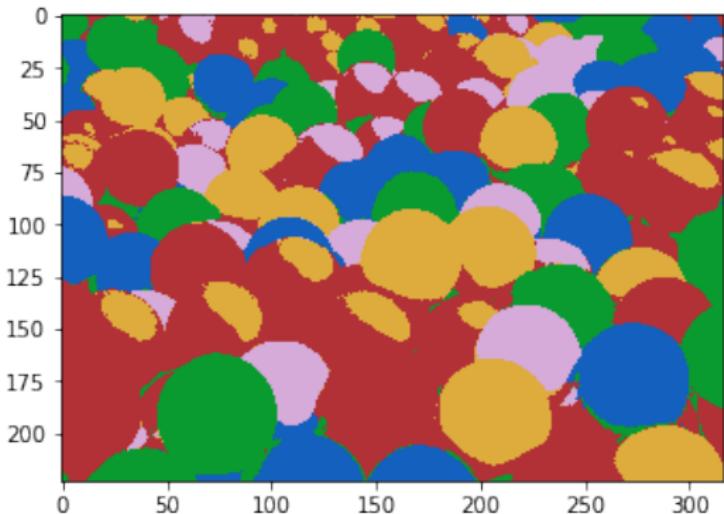
- ① Wybór palety  $K$  kolorów, dostosowanej do obrazka.
- ② Sklejanie obrazka z kwadratów (kompresja wektorowa)
- ③ Grupowanie słów podobnych (jeżeli reprezentujemy słowa jako wektory)
  - Po zastosowaniu do fraz: [odpoczynek w pięknym miasteczku](#)  
 $\approx$  [wypoczynek w uroczym kurorcie](#)

# $K$ -średnik dla kolorów

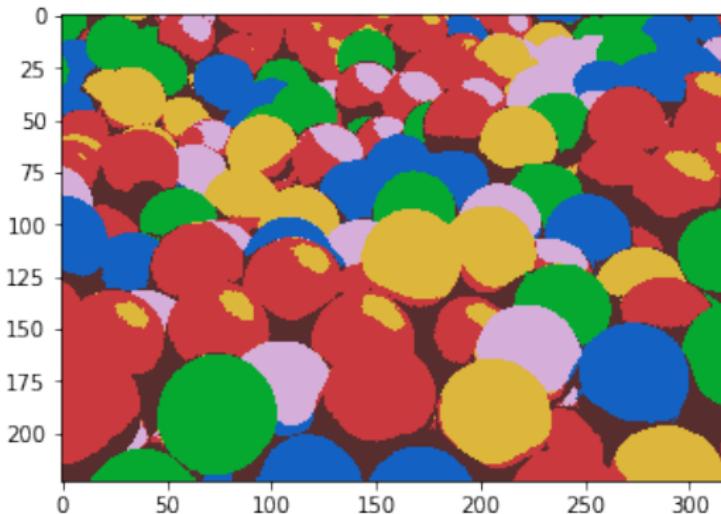
Popatrzmy, jak wybór palety i kompresja wektorowa działają dla przykładowego obrazka:



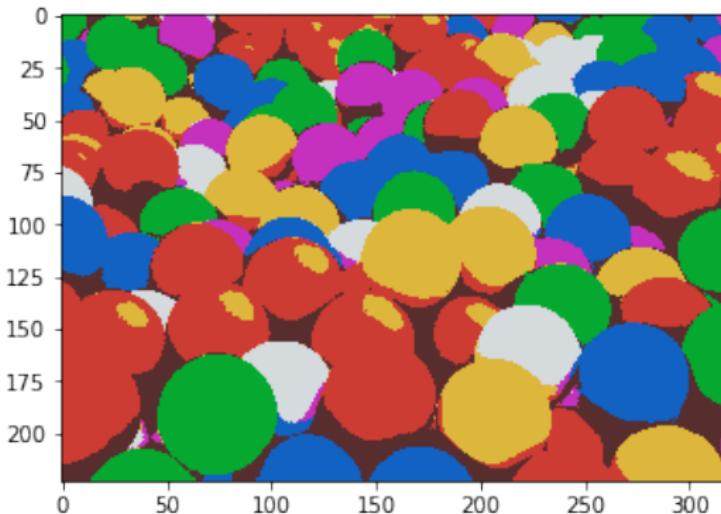
# $K$ -średnich. Wybór reprezentatywnych pikseli, $K=5$



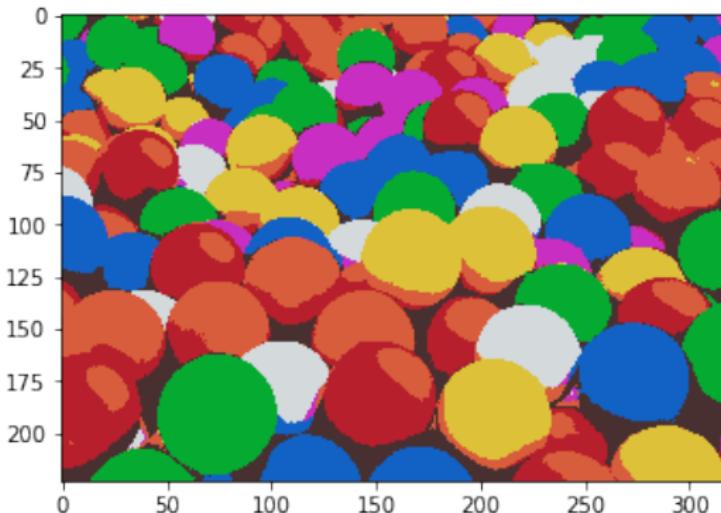
# $K$ -średnich. Wybór reprezentatywnych pikseli, $K=6$



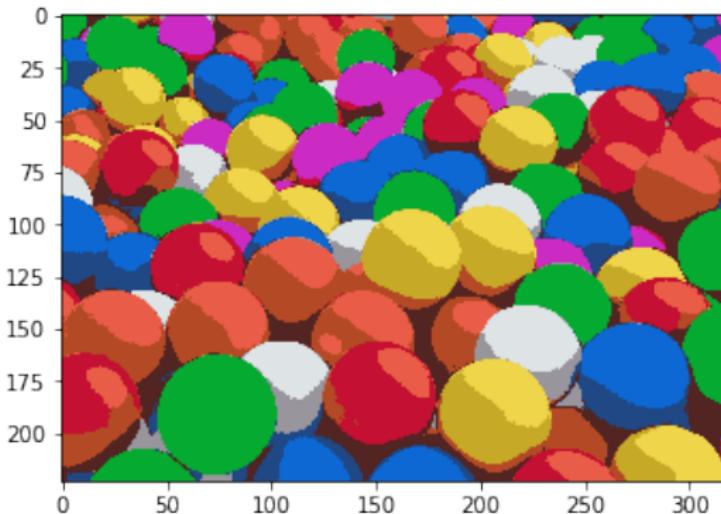
# $K$ -średnich. Wybór reprezentatywnych pikseli, $K=7$



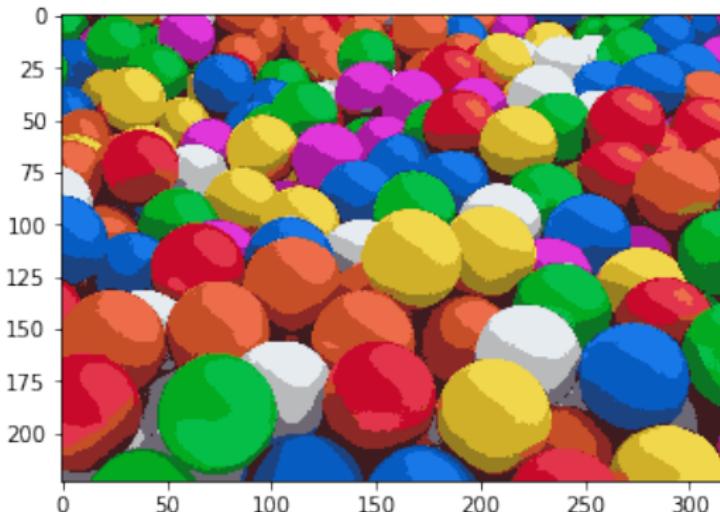
# $K$ -średnich. Wybór reprezentatywnych pikseli, $K=8$



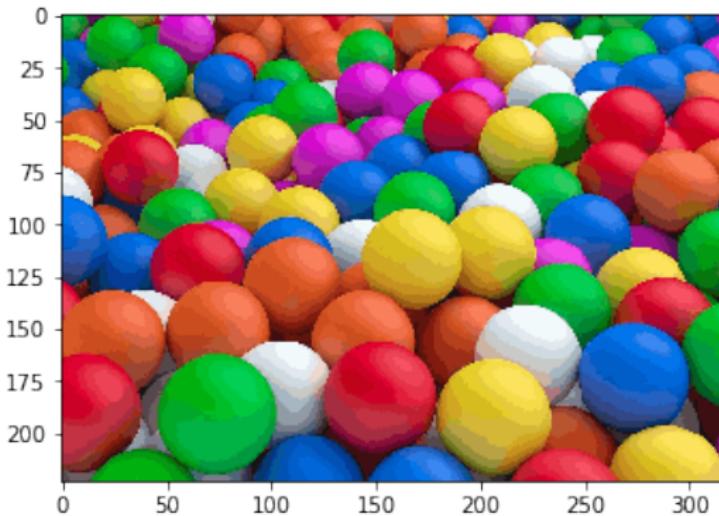
# $K$ -średnich. Wybór reprezentatywnych pikseli, $K=12$



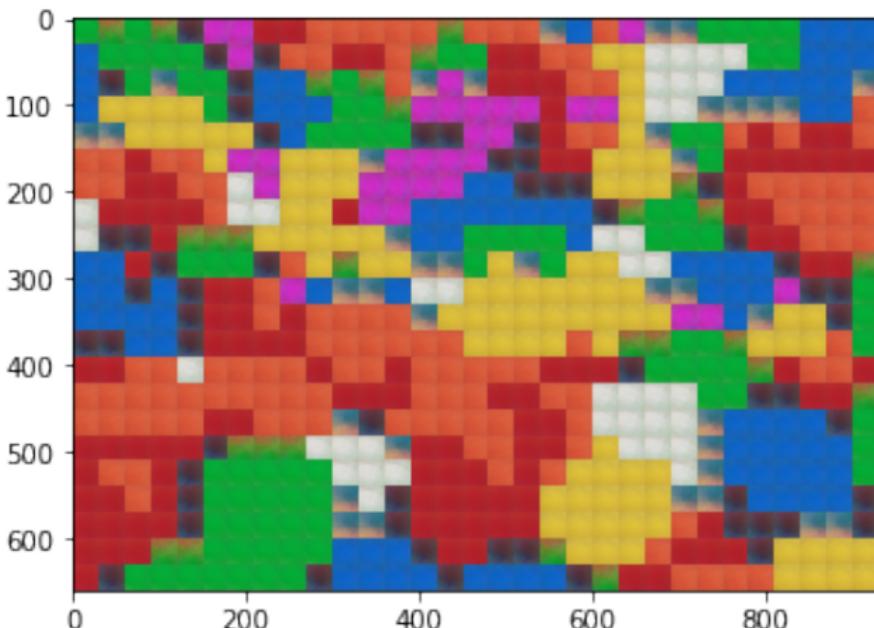
# $K$ -średnich. Wybór reprezentatywnych pikseli, $K=20$



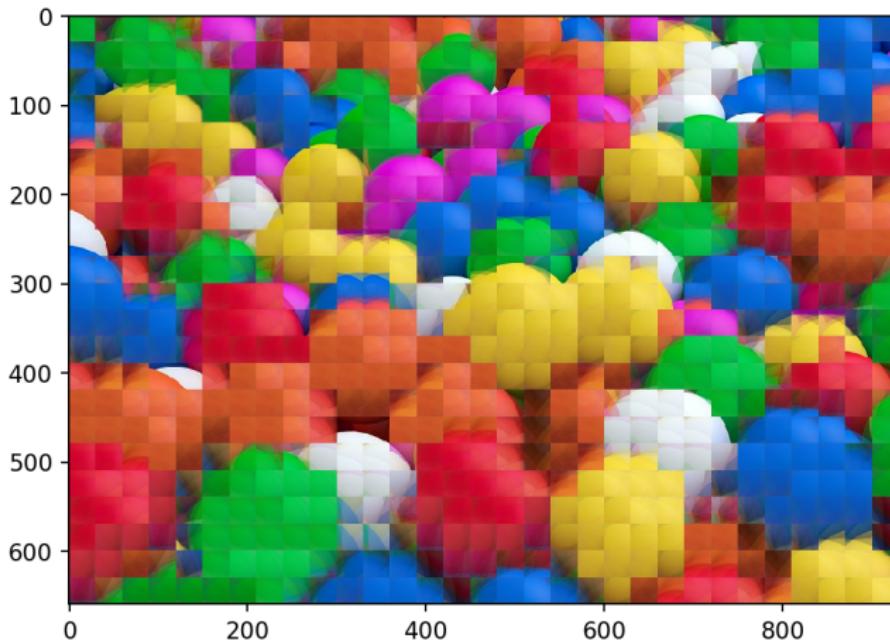
# $K$ -średnich. Wybór reprezentatywnych pikseli, $K=100$



# Na deser – wynik kompresji wektorowej, K=10



# Na deser – wynik kompresji wektorowej, K=100



# K-średnich. Kilka uwag końcowych

- Możemy powtarzać losowanie punktów kilka razy i wybrać najmniejszy błąd
- Możemy wykonać (w celu przyspieszenia) algorytm dla pod populacji punktów (i potem tylko 1-2 etapy dla wszystkich punktów)
- Zauważmy, że im większe  $K$ , tym (średnio) mniejszy błąd grupowania

## Pytanie

Jak wykorzystać ostatnie spostrzeżenie do wyboru wartości  $K$   
(wskazówka: jak wygląda wykres wartości błędu w zależności od  $K$ )

# Błąd w zależności od liczby klastrów

- 1 klaster – **maksymalny błąd**
- $N$  klastrów – **minimalny błąd** (każdy swoim reprezentantem)
- $K$  jest mniejsze od naturalnej liczby skupień – pewne klastry są połączeniem wielu, duży błąd

## Uwaga

Gdy zwiększamy  $K$ , do pewnego momentu błąd maleje znacznie, od któregoś – krzywa się wypłaszcza

**ten moment to rzeczywista liczba skupień**

# Wykrywanie nieprawidłowości

Potencjalnie bardzo użyteczne zadanie: można na przykład analizować pomiary różnych parametrów jakiegoś skomplikowanego systemu (skrzydło samolotu pasażerskiego) i zauważać, że coś dziwnego się dzieje

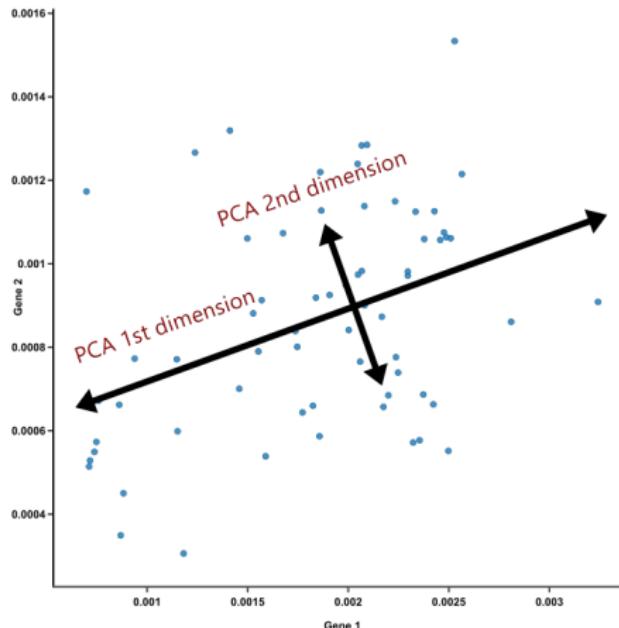
- **Pytanie:** Jak wykorzystać algorytm K-means do wykrywania anomalii?
- Charakterystyka anomalii:
  - Daleko od centrum
  - Najbliżsi sąsiedzi należą do różnych klastrów

Wykonanie algorytmu  $K$ -średnich i analiza poszczególnych punktów daje możliwość zidentyfikowania „dziwnych” elementów (potencjalnych nieprawidłowości).

- **Cel:** „zagęszczanie danych” (umożliwiające, być może, lepsze działanie innych algorytmów)
- **Dodatkowa korzyść:** jak zredukujemy liczbę wymiarów do 2, to możemy zbiór danych ładnie narysować (i być może zobaczyć jakieś prawidłowości)
- Redukcja wymiarów oznacza usunięcie informacji, ale aby móc usunąć **nieistotne informacje**, czyli szum.
- Przykładową metodą (omawianą na algebrze) jest **PCA**, czyli **analiza głównych składowych**

# Principal Component Analysis

- Identyfikujemy osie, które odpowiadają za największą zmienność danych
- Obracamy przestrzeń i pozostawiamy tylko najważniejsze wymiary.



- Mamy punkty w przestrzeni wielowymiarowej.
- Chcemy przypisać im punkty na płaszczyźnie (2D)
- Jak? (wskazówka: potrafimy liczyć odległość w przestrzeni wielowymiarowej)

## Ogólna zasada

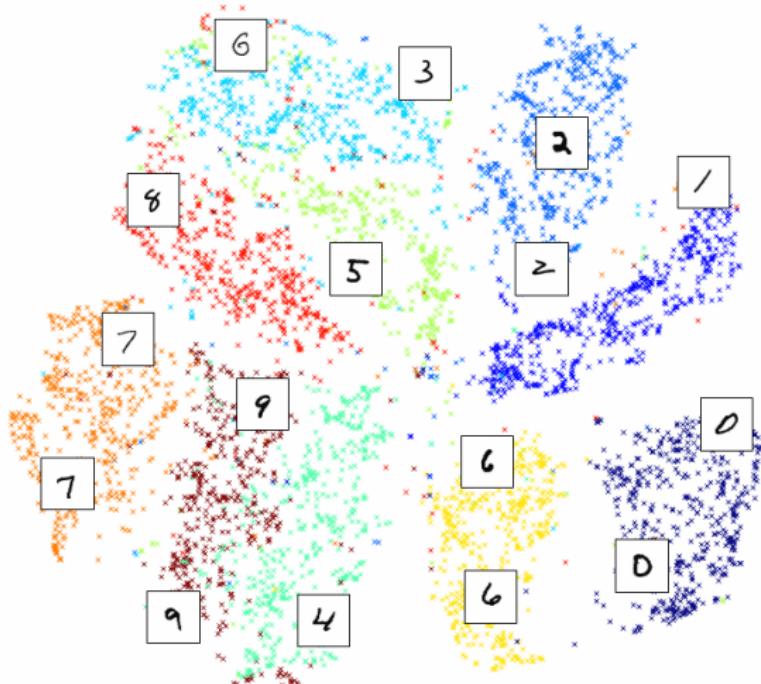
Staramy się, by odległości w 2D odpowiadały tym w oryginalnej przestrzeni (np. 500D)

Niektóre algorytmy można interpretować jako tworzenie układu punktów połączonych sprężynkami, punkty „podobne” się przyciągają, odległe – odpychają, szukamy równowagi tego układu dynamicznego.

# Wizualizacja obrazów cyfr (MNIST)

Algorytm t-SNE (który można interpretować „sprzęzynkowo”):

MNIST dataset – Two-dimensional embedding of 70,000 handwritten digits with t-SNE



# Koniec części II

- Definiujemy

$$\hat{Q}_{\text{opt}}(s, a; \mathbf{w}) = \mathbf{w} \cdot \phi(s, a)$$

- Weźmy grę w Dżunglę z losowym przeciwnikiem (zauważmy, że to jest MDP)
- Przykładowe cechy (propozycje?):
  - Czy jest bicie (0/1)?
  - Czy jest bicie słonia, szczura, kota (dla każdej bierki cecha)?
  - Czy ruch zbliża do jamy przeciwnika? (0/1)
  - Czy ruch jest skokiem zbliżającym do jamy?
  - Czy najbliższa jamie bierka się zbliżyła?
  - Czy podchodzimy pod bicie?
  - Czy bijemy bierkę w pułapce
  - Czy ruch jest do przodu (w lewo, w prawo, w dół)?
  - Czy zajmujemy któryś z wskazanych pól (związanych z atakiem bądź obroną)

- **Temporal Difference Learning** – metoda uczenia wartości  $V$  (używana na przykład w grach)
- Idea:
  - Generuj dane z rozgrywek
  - Naucz się wag funkcji heurystycznej analizując te dane

## Uwaga

W najprostszym przypadku (czyli liniowym) mamy:

$$V(s; w) = w \cdot \phi(s)$$

## Definicja

**Polityka odruchów (reflex policy)** – to strategia agenta, w której podejmuje decyzje analizując przybliżoną funkcję oceniającą konsekwencje działań (w grach: stany po ruchu)

- Do generowania danych możemy wykorzystać politykę odruchową (czyli naszą aktualną funkcję oceniającą)
- **Problem:** tak wygenerujemy tylko jedną rozgrywkę (albo bardzo niezróżnicowaną populację rozgrywek)

## Generowanie danych (2)

Konieczne jest wprowadzenie losowości:

- a) Polityka  $\varepsilon$ -zachłanna (pamiętajmy o zmianie znaczenia  $V$  dla Mina i Maxa)
- b) Losowanie zgodne z prawdopodobieństwem „softmaxowym”, czyli:

$$P(s, a) = \frac{e^{V(\text{succ}(s, a))}}{\sum_{a' \in \text{Actions}(s)} e^{V(\text{succ}(s, a'))}}$$

- c) Dla gier z rzucaniem kostkami (z elementem losowym) można wybierać zawsze optymalne ruchy (sama gra zapewnia czynnik eksploracyjny)

- **Predykcja:**  $V(s; w)$
- **Cel:**  $r + \gamma V(s'; w)$  ( $s'$  to kolejny stan w rozgrywce)

## Streszczenie reguły

W przypadku większości gier ( $\gamma = 1$ , nagroda na końcu), sprowadza się to do:

- Staraj się, by podczas dobrych gier wartość planszy po ruchu zbytnio się nie zmieniała (dla minmaxa i optymalnej strategii powinna być ona stała)
- Zwiększaj wartość sytuacji bliskich zwycięstwa (wykorzystanie  $r$  pod koniec).

- Funkcja celu:

$$\frac{1}{2}(\text{prediction}(w) - \text{target})^2$$

- Gradient:

$$(\text{prediction}(w) - \text{target})\nabla_w(\text{prediction}(w))$$

- Reguła uaktualniania:

$$w \leftarrow w - \eta((\text{prediction}(w) - \text{target})\nabla_w(\text{prediction}(w)))$$

## Algorytm

Dla każdego  $s, a, r, s'$  wykonuj:

$$w \leftarrow w - \eta(V(s; w) - (r + \gamma V(s', w))) \nabla_w V(s; w)$$

Dla funkcji liniowej:

$$V(s, w) = w \cdot \phi(s)$$

mamy

$$\nabla_w V(s, w) = \phi(s)$$

# Porównanie TD-learning i Q-learning



## Algorithm: TD learning

On each  $(s, a, r, s')$ :

$$\mathbf{w} \leftarrow \mathbf{w} - \eta [\underbrace{\hat{V}_\pi(s; \mathbf{w})}_{\text{prediction}} - \underbrace{(r + \gamma \hat{V}_\pi(s'; \mathbf{w}))}_{\text{target}}] \nabla_{\mathbf{w}} \hat{V}_\pi(s; \mathbf{w})$$



## Algorithm: Q-learning

On each  $(s, a, r, s')$ :

$$\mathbf{w} \leftarrow \mathbf{w} - \eta [\underbrace{\hat{Q}_{\text{opt}}(s, a; \mathbf{w})}_{\text{prediction}} - \underbrace{(r + \gamma \max_{a' \in \text{Actions}(s)} \hat{Q}_{\text{opt}}(s', a'; \mathbf{w}))}_{\text{target}}] \nabla_{\mathbf{w}} \hat{Q}_{\text{opt}}(s, a; \mathbf{w})$$

- Można łączyć TD learning z uczeniem polityki. AlphaGo Zero tak właśnie robiło.
- Można stosować metody **poprawiania polityki/oceny**:
  - a) Bazujące na alpha-beta search (tak uczyć funkcję ocenającą, żeby miała „inteligencję” taką, jak poprzednia wersja)
  - b) MCTS policy improvement (żeby nowa polityka udawała jak najlepiej starą wspomagającą się symulacjami MCTS)

## Uwaga

Takie metody były używane w różnych słynnych programach:

1. Warcaby (Samuel, 1965)
2. Tryktrak, czyli Backgammon (Tesauro, ok. 1990)
3. AlphaGoZero (DeepMing, 2017)

- AlphaGo wygrało z Lee Sedolem (drugi gracz na świecie).

- **AlphaGo** wygrało z Lee Sedolem (drugi gracz na świecie). Zawierało elementy uczenia z historycznych partii – uczenie **z nadzorem**, jaki ruch zrobił dobry gracz, który w sytuacji X wygrał partię.
- **AlphaZero** – uczył się tylko grając sam ze sobą!
- Bardzo dobre (lub znakomite) wyniki w grach:
  - Go (mistrz Wszechświata)
  - Szachy (ale nie poziom mistrza świata)
  - Shoggi (jako przykład innej gry, która też działa)

## Chess



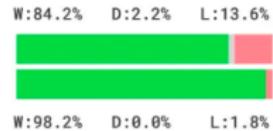
AlphaZero vs. Stockfish



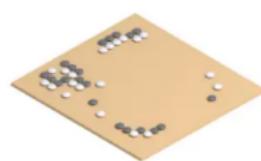
## Shogi



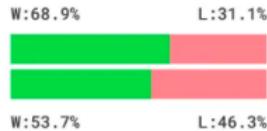
AlphaZero vs. Elmo



## Go



AlphaZero vs. AGO



AZ wins



AZ draws



AZ loses



AZ white ○



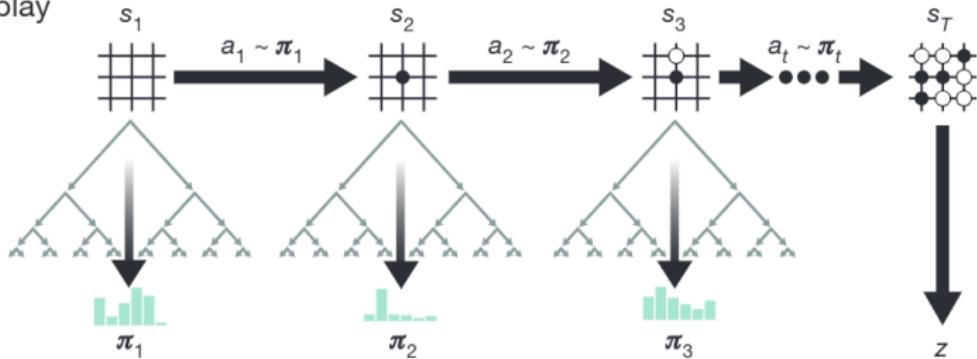
AZ black ●



<https://deepmind.com/blog/article/alphazero-shedding-new-light-grand-games-chess-shogi-and-go>

# AlphaZero. Rozgrywka

## a Self-play



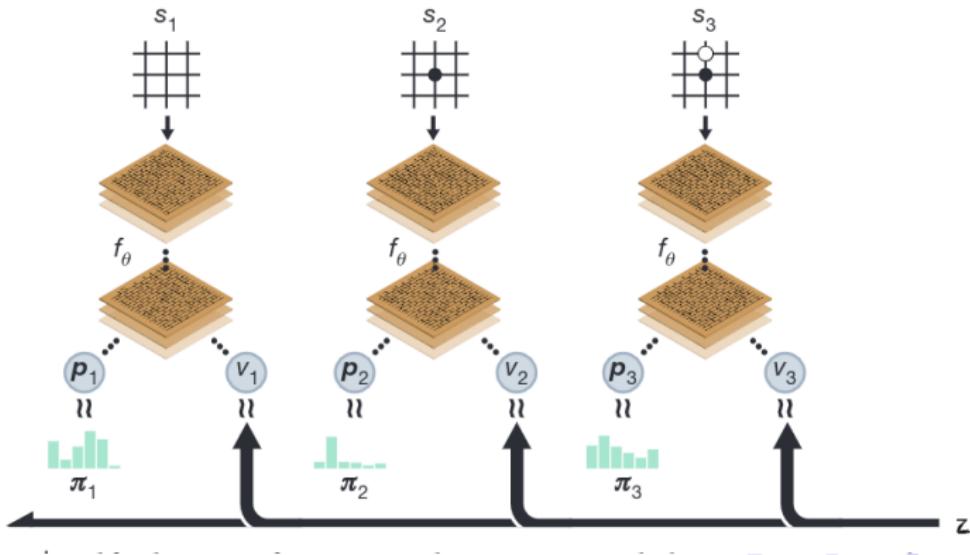
## Drobna uwaga

Trochę zmodyfikowany MCTS, który zamiast symulacji używa aktualnej funkcji heurystycznej.

## Cele uczenia

- 1 Funkcja wybierająca ruch
- 2 Funkcja oceniająca planszę

### b Neural network training



# Logika

Paweł Rychlikowski

Instytut Informatyki UWr

29 maja 2021

- ① Bazujące na **stanach**: przeszukiwanie, MDP, gry
- ② Bazujące na **zmiennych**: CSP, sieci Bayesowskie (jeszcze przed nami)
- ③ Bazujące na **logice**: logika zdaniowa, logiki modalne, logika 1-go rzędu

Zajmiemy się teraz logiką. Zaczniemy od **logiki zdaniowej**.

- zmienne zdaniowe (przyjmują wartości 0/1)
- spójniki:  $\vee$ ,  $\wedge$ ,  $\rightarrow$ ,  $\leftrightarrow$ ,  $\neg$

## Przykłady

- pada  $\wedge \neg$  mam-parasol  $\rightarrow$  jestem-mokry  $\vee$  mam-kurtkę
- $\neg(p \vee q) \leftrightarrow (\neg p \wedge \neg q)$

- **Modelem** w logice zdaniowej jest przypisanie zmiennym wartości logicznych.
  - Ogólnie o modelu myślimy jako o naszej wizji świata.
- **Interpretacją** formuły przy zadanym modelu jest zdefiniowana rekurencyjnie **wartość** formuły:
  - $I(a, w) = w(a)$ , jeżeli  $a$  jest zmienną
  - $I(f_1 \vee f_2, w) = I(f_1, w) \vee I(f_2, w)$
  - (...) – inne, podobne reguły
- Składnia (syntax) vs semantyka

## Definicja

Formuła  $f$  jest **spełnialna**, czyli ma model, jeżeli istnieje takie  $w$ , że  $I(f, w) = 1$ .

## Uwaga

Takich przypisań jest skończenie wiele, stąd mamy prosty (wykładowczy) algorytm sprawdzania, czy formuła ma model (**jest spełnialna**)

- Formuły to zdania opisujące świat.
- Naturalne jest myślenie o zbiorze takich formuł (do którego możemy dodawać nowe fakty).
- Taki zbiór często nazywamy **bazą wiedzy**.

## Definicje

1. literał - zmienna albo  $\neg$  zmienna
2. klauzula -  $I_1 \vee \dots \vee I_n$  (gdzie  $I_i$  to literał)
3. formuła w CNF -  $c_1 \wedge \dots \wedge c_n$ , gdzie  $c_i$  jest klauzulą

# Dlaczego CNF jest fajna?

- Każdą formułę można przekształcić do CNF (czasem płacąc wykładowniczym wzrostem jej długości, ćwiczenia)
- Koniunkcja klauzul = zbiór klauzul = baza wiedzy
- Jak mamy zbiór formuł (bazę wiedzy, niekoniecznie w CNF) to wykładowniczość dotyczy pojedynczej formuły, a nie całej bazy.

- Sprawdzanie spełnialności formuły boolowskiej jest zadaniem rozwiązywania więzów (baza wiedzy = zbiór więzów)
- Nie dziwi zatem, że podstawowe algorytmy (stosowane w praktyce) są dość podobne ([backtracking + propagacja](#)).

## Uwaga

Współczesne **SAT-solvery** radzą sobie z milionami klauzul i setkami tysięcy zmiennych

# A NP-zupełność?

- Oczywiście problem CNF-SAT (spełnialności formuły w CNF) jest **NP-zupełny**.
- Nie spodziewamy się istnienia algorytmu wielomianowego (znane algorytmy mają pesymistyczny czas wykładniczy).

## Pytanie

Dlaczego SAT-Solvery działają dobrze?

Pytanie jest trudne, i tak do końca nie ma odpowiedzi. Jedyne, co można powiedzieć, że widocznie znaczna część w praktyce spotykanych formuł jest w jakimś sensie **łatwa**.

- Algorytm **Davisa–Putnama–Logemanna–Lovelanda (DPLL)** jest algorytmem znajdującym spełniające podstawienie dla formuły CNF.
- Jest **zupełny** – tzn. zawsze kończy się z prawidłowym wynikiem, może działać dłujo

## Uwaga

Stanowi bazę współczesnych SAT-Solverów (z jednym usprawnieniem, o którym jeszcze powiemy).

## Definicje

- a) **Klauzula jednostkowa** (unit clause) – klauzula zwierająca 1 literał
- b) **Czysty literał** – literał, który występuje tylko jako pozytywny, lub tylko jako negatywny (czyli z jedną polaryzacją).

$$x_1 \wedge \neg x_2 \wedge (x_3 \vee x_4 \vee x_5) \wedge (\neg x_3 \vee \neg x_4 \vee x_5)$$

Jakie wnioskowanie można przeprowadzić korzystając z tych pojęć:

- a) **unit propagation** – klauzule jednostkowe można spełnić na 1 sposób (spełniając literał), wstawiając wartość logiczną do innych klauzul możemy zrobić nowe klauzule jednostkowe.
- b) „Opłaca się” przypisywać **czystym literałom** wartość **true** (bo?).

## Algorytm

Funkcja **DPLL**( $\Phi$ ):

- jeśli  $\Phi$  zawiera pustą klauzulę zwróć **false**
- dla każdej klazuli jednostkowej wykonaj **unit propagation** zmieniając  $\Phi$  (do nasycenia)
- ustal wartości dla czystych literałów (zmieniając  $\Phi$ )
- wybierz zmienną  $x$  (o nieokreślonej do tej pory wartości)
- zwróć **DPLL**( $\Phi \wedge x$ ) **or** **DPLL**( $\Phi \wedge \neg x$ )

Oczywiście czasem wystarczy sprawdzić tylko jedną część rekurencyjnego wywołania!

- Zauważmy, że jak w algorytmie DPLL osiągnęliśmy sprzeczność, to można myśleć, że „udowodniliśmy” twierdzenie (przy założeniu analizowanej formuły  $\Phi$ ):

$$(l_1 \wedge l_2 \wedge \dots \wedge l_n) \rightarrow \text{Fałsz}$$

gdzie  $l_i$  jest literałem użyтыm w  $i$ -tym momencie algorytmu (w propagacji lub w backtrackingu)

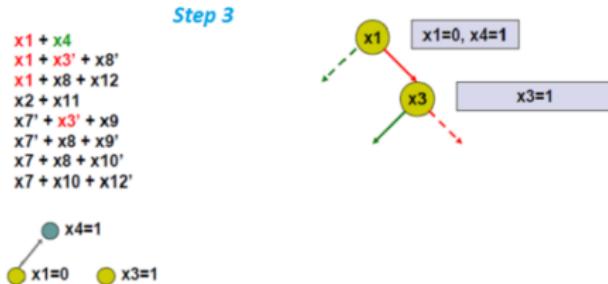
- Możemy zatem uznać, że udowodniliśmy twierdzenie: ze zbioru klauzul wynika  $\neg l_1 \vee \dots \vee \neg l_n$
- Trochę długa formuła (i nieużyteczna w propagacji), algorytm próbuje zatem znaleźć możliwie krótki ciąg literałów o ustalonych wartościach, który sam z siebie implikuje Fałsz (i zapamiętać go na przyszłość)

# Conflict-driven clause learning (CDCL)

W stosunku do DPLL mamy dwie istotne modyfikacje:

- ① Po dojściu do sprzeczności możemy dodać nową klauzulę, która **podsumowuje przyczynę sprzeczności**
- ② Przy nawrocie możemy cofnąć się do wcześniejszej zmiennej (**„praprzyczyny sprzeczności”**)

# Przykład działania CDCL



- Przeanalizujemy zaczerpnięty z Wikipedii przykład działania CDCL.
- Notacja:
  - Mamy literały: **niezwartościowane**, **pozytywne** oraz **negatywne**.
  - Mamy graf implikacji, w którym zapisujemy, jakie konsekwencje powodują nasze wybory
  - Wybory „dowolne” są brudnożółte.

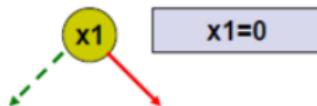
## Uwaga

Dla czytelności przykład nie uzgęldnia obsługi **czystych literałów!**

# Przykład działania CDCL z Wikipedii

$x_1 + x_4$   
 $x_1 + x_3' + x_8'$   
 $x_1 + x_8 + x_{12}$   
 $x_2 + x_{11}$   
 $x_7' + x_3' + x_9$   
 $x_7' + x_8 + x_9'$   
 $x_7 + x_8 + x_{10}'$   
 $x_7 + x_{10} + x_{12}'$

*Step 1*

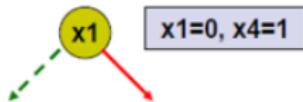


●  $x_1 = 0$

# Przykład działania CDCL z Wikipedii

## Step 2

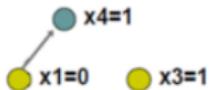
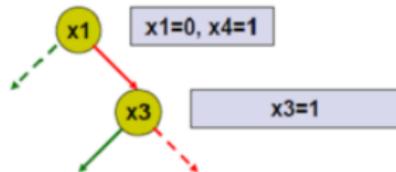
$x_1 + x_4$   
 $x_1 + x_3' + x_8'$   
 $x_1 + x_8 + x_{12}$   
 $x_2 + x_{11}$   
 $x_7' + x_3' + x_9$   
 $x_7' + x_8 + x_9'$   
 $x_7 + x_8 + x_{10}'$   
 $x_7 + x_{10} + x_{12}'$



# Przykład działania CDCL z Wikipedii

## Step 3

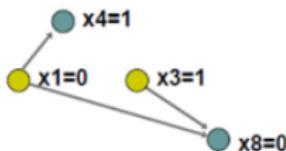
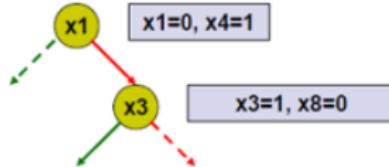
$x_1 + x_4$   
 $x_1 + x_3' + x_8'$   
 $x_1 + x_8 + x_{12}'$   
 $x_2 + x_{11}$   
 $x_7' + x_3' + x_9$   
 $x_7' + x_8 + x_9'$   
 $x_7 + x_8 + x_{10}'$   
 $x_7 + x_{10} + x_{12}'$



# Przykład działania CDCL z Wikipedii

## Step 4

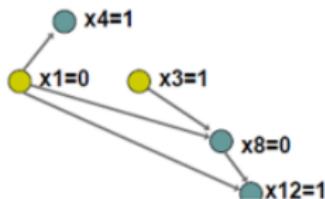
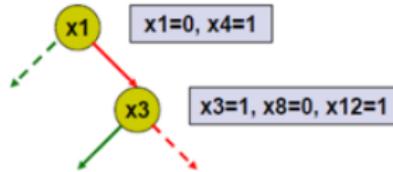
$x_1 + x_4$   
 $x_1 + x_3' + x_8'$   
 $x_1 + x_8 + x_{12}$   
 $x_2 + x_{11}$   
 $x_7' + x_3' + x_9$   
 $x_7' + x_8 + x_9'$   
 $x_7 + x_8 + x_{10'}$   
 $x_7 + x_{10} + x_{12'}$



# Przykład działania CDCL z Wikipedii

$x_1 + x_4$   
 $x_1 + x_3' + x_8'$   
 $x_1 + x_8 + x_{12}$   
 $x_2 + x_{11}$   
 $x_7' + x_3' + x_9$   
 $x_7' + x_8 + x_9'$   
 $x_7 + x_8 + x_{10}'$   
 $x_7 + x_{10} + x_{12}'$

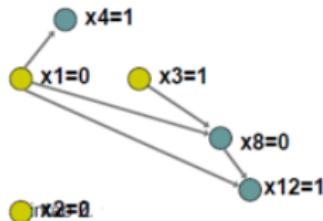
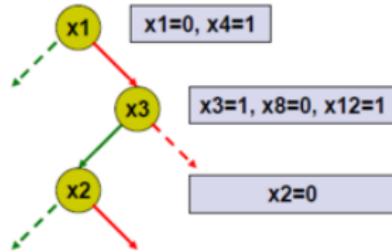
## Step 5



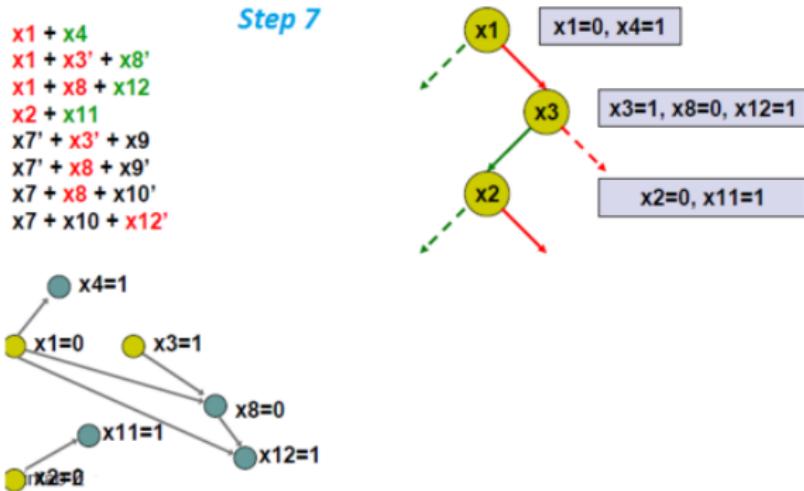
# Przykład działania CDCL z Wikipedii

$x_1 + x_4$   
 $x_1 + x_3' + x_8'$   
 $x_1 + x_8 + x_{12}'$   
 $x_2 + x_{11}$   
 $x_7' + x_3' + x_9$   
 $x_7' + x_8 + x_9'$   
 $x_7 + x_8 + x_{10}'$   
 $x_7 + x_{10} + x_{12}'$

*Step 6*



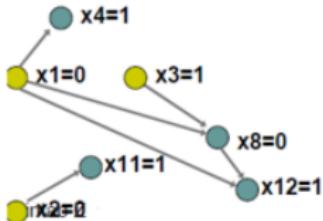
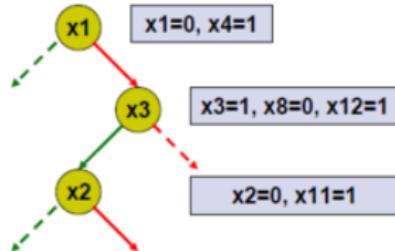
# Przykład działania CDCL z Wikipedii



# Przykład działania CDCL z Wikipedii

$x_1 + x_4$   
 $x_1 + x_3' + x_8'$   
 $x_1 + x_8 + x_{12}'$   
 $x_2 + x_{11}$   
 $x_7' + x_3' + x_9$   
 $x_7' + x_8 + x_9'$   
 $x_7 + x_8 + x_{10}'$   
 $x_7 + x_{10} + x_{12}'$

*Step 8*

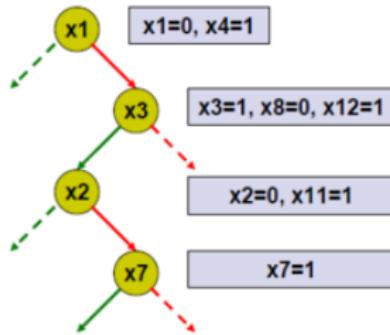
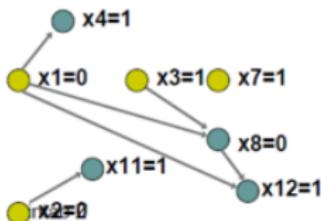


Koniec części I

# Przykład działania CDCL z Wikipedii

$x_1 + x_4$   
 $x_1 + x_3' + x_8'$   
 $x_1 + x_8 + x_{12}$   
 $x_2 + x_{11}$   
 $x_7' + x_3' + x_9$   
 $x_7' + x_8 + x_9'$   
 $x_7 + x_8 + x_{10}'$   
 $x_7 + x_{10} + x_{12}'$

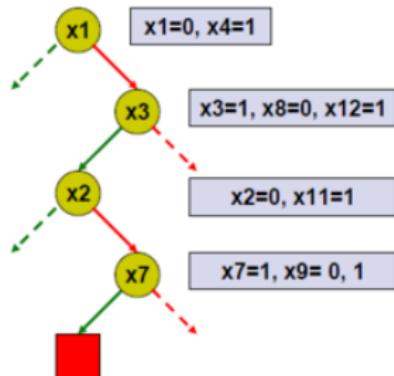
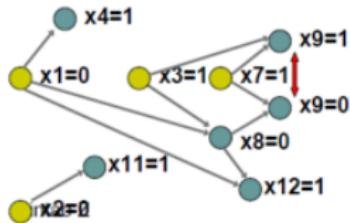
*Step 9*



# Przykład działania CDCL z Wikipedii

$x_1 + x_4$   
 $x_1 + x_3' + x_8'$   
 $x_1 + x_8 + x_{12}$   
 $x_2 + x_{11}$   
 $x_7' + x_3' + x_9$   
 $x_7' + x_8 + x_9'$   
 $x_7 + x_8 + x_{10}'$   
 $x_7 + x_{10} + x_{12}'$

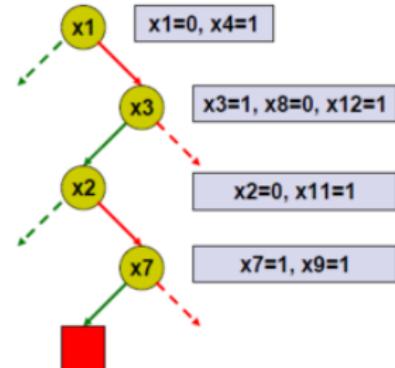
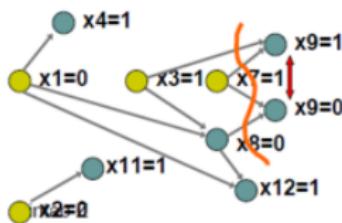
*Step 10*



# Przykład działania CDCL z Wikipedii

$x_1 + x_4$   
 $x_1 + x_3' + x_8'$   
 $x_1 + x_8 + x_{12}'$   
 $x_2 + x_{11}$   
 $x_7' + x_3' + x_9$   
 $x_7' + x_8 + x_9'$   
 $x_7 + x_8 + x_{10}'$   
 $x_7 + x_{10} + x_{12}'$

Step 11



$x_3 = 1 \wedge x_7 = 1 \wedge x_8 = 0 \rightarrow \text{conflict}$

# Przykład działania CDCL z Wikipedii

If  $a$  implies  $b$ , then  $b'$  implies  $a'$

**Step 12**

$x_3=1 \wedge x_7=1 \wedge x_8=0 \rightarrow$  conflict

Not conflict  $\rightarrow (x_3=1 \wedge x_7=1 \wedge x_8=0)'$

true  $\rightarrow (x_3=1 \wedge x_7=1 \wedge x_8=0)'$

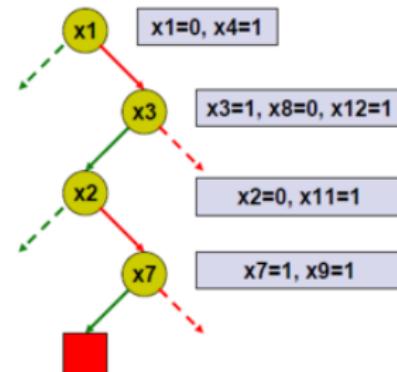
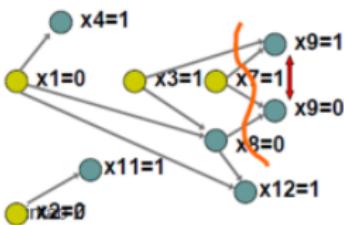
$(x_3=1 \wedge x_7=1 \wedge x_8=0)'$

$(x_3' + x_7' + x_8)$

# Przykład działania CDCL z Wikipedii

$x_1 + x_4$   
 $x_1 + x_3' + x_8'$   
 $x_1 + x_8 + x_{12}$   
 $x_2 + x_{11}$   
 $x_7' + x_3' + x_9$   
 $x_7' + x_8 + x_9'$   
 $x_7 + x_8 + x_{10}'$   
 $x_7 + x_{10} + x_{12}'$

*Step 13*



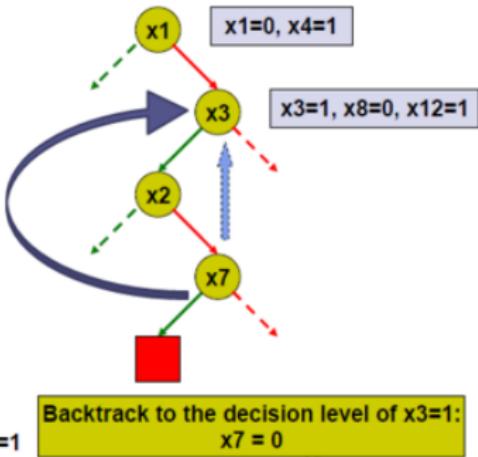
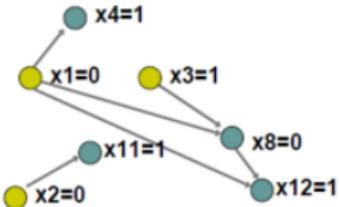
$x_3 = 1 \wedge x_7 = 1 \wedge x_8 = 0 \rightarrow \text{conflict}$

Add conflict clause:  $x_3' + x_7' + x_8$

# Przykład działania CDCL z Wikipedii

$x_1 + x_4$   
 $x_1 + x_3' + x_8'$   
 $x_1 + x_8 + x_{12}'$   
 $x_2 + x_{11}$   
 $x_7' + x_3' + x_9$   
 $x_7' + x_8 + x_{9'}$   
 $x_7 + x_8 + x_{10'}$   
 $x_7 + x_{10} + x_{12'}$   
 $x_3' + x_8 + x_7'$

Step 14



- Trzeba zarządzać „nowymi” klauzulami, monitorować ich przydatność, może kasować...

## Uwaga

Obecnie jest to najbardziej efektywna metoda testowania spełnialności (i znajdowania podstawienia).

- Wypada coś powiedzieć o drugim (również używanym) algorytmie, którym jest ...

**WalkSAT**

1. Zaczynamy od losowego przypisania zmiennym wartości logicznych.
2. Jak wszystkie klauzule są spełnione (mają co najmniej 1 pozytywny literał), to **koniec**
3. Wybierz losową klauzulę, która jest niespełniona
4. Rzuć monetą (prawdopodobieństwo  $p$ ):
  - a) Orzeł: zmień wartość jednej zmiennej z klauzuli (**teraz jest spełniona!**)
  - b) Reszka: zmień wartość tej zmiennej z klauzuli, która maksymalizuje: **różnicę klauzul spełnionych i niespełnionych**
5. Po określonej liczbie zmian można zrobić **restart**, ewentualnie zwrócić stałą **porażka**.

- ① **PLUS:** Jak zakończy działanie z sukcesem, to formuła jest spełnialna (i znaleźliśmy podstawienie)
- ② **PLUS:** Mamy pełną kontrolę nad czasem działania
- ③ **MINUS:** Nie możemy mówić o **niespełnialności**: porażka nie oznacza.

- Jak już mówiliśmy, nie jest to w pełni satysfakcjonująco rozwiążane.
- Przedstawimy parę spostrzeżeń o formułach losowych

## Uwaga

Formuły w CNF możemy łatwo parametryzować. Mają one prostą strukturę, daną przez:

- Liczbę różnych zmiennych
- Liczbę klauzul
- Maksymalną wielkość klauzuli

Rozważmy prawdopodobieństwo spełnialności formuły 3-CNF, w zależności od **liczby klauzul** oraz **liczby zmiennych**.

- Dużo zmiennych –
- Dużo klauzul –

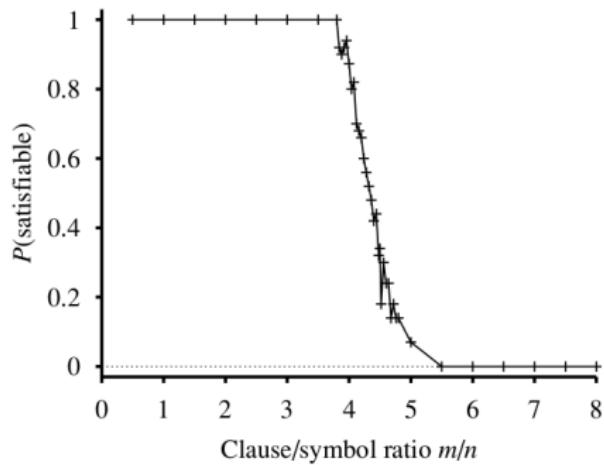
Rozważmy prawdopodobieństwo spełnialności formuły 3-CNF, w zależności od **liczby klauzul** oraz **liczby zmiennych**.

- Dużo zmiennych – **łatwo spełnialna**
- Dużo klauzul –

Rozważmy prawdopodobieństwo spełnialności formuły 3-CNF, w zależności od **liczby klauzul** oraz **liczby zmiennych**.

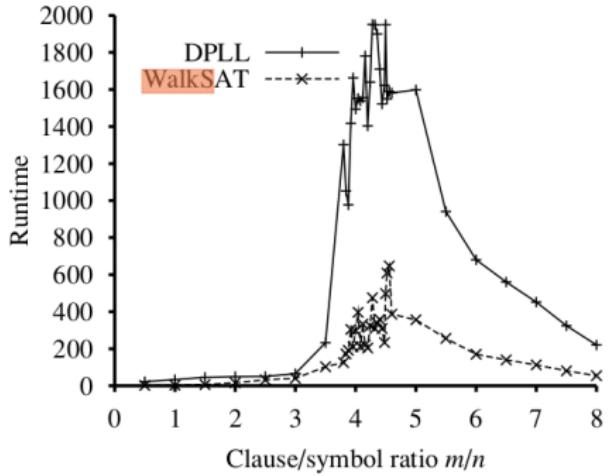
- Dużo zmiennych – **łatwo spełnialna**
- Dużo klauzul – **trudno spełnialna**

# Losowe CNF. Prawdopodobieństwo spełnienia



- Dużo zmiennych: może wiele z jedną polaryzacją?
- Dużo zmiennych: może dłuższe klauzule (a do spełnienia wystarczy 1 literał)
- Dużo klauzul – a każda musi być spełniona

# Losowe CNF. Czas trwania



Punkt krytyczny w okolicy  $\mathbf{m/n = 4.3}$

# Wykorzystanie logiki zdaniowej

- Zadania z więzami realizowane za pomocą logiki zdaniowej  
**Przykład:** obrazki **logiczne**
- Algorytmy planowania  
Przeszukiwanie specjalnej przestrzeni stanów, w której dozwolone ruchy opisane są **formułami**
- Agenty hybrydowe (przeplatanie wnioskowania z innymi metodami)

## Przykładowa reprezentacja

- Mamy zmienne (binarne) odpowiadające polom,
- Mamy zmienne typu: w wierszu 5 jest układ 00111001100

Co dalej?

## Formuły

- W każdym wierszu (bądź kolumnie) jest jeden ze zgodnych ze specyfikacją układów (długa alternatywa)
- Formuły „tłumaczące” zmienne wierszowe na zmienne związane z polami:

$$W_{01101110} \rightarrow \neg X_0 \wedge X_1 \wedge X_2 \wedge \neg X_3 \wedge X_4 \wedge X_5 \wedge X_6$$

(to jest skrót dla  $n$  formuł typu  $W_{01101110} \rightarrow L_i$ )

## Uwaga

Mamy czas, czyli  $t \in 0, \dots, T - 1$ . Używamy zmiennych mówiących o stanie świata w momencie  $t$  i o akcji w momencie  $t$ . Opisujemy mechanikę świata językiem logiki.

- Określamy zmienne prawdziwe w czasie 0,  
 $\text{stoję-przed-sklepem}_0 \wedge \text{pusta-torba}_0 \wedge \text{pełen-portfel}_0 \wedge \dots$
- Określamy cel (czyli co chcemy uzyskać w czasie  $T$ ) – czas  $T$  musimy zgadnąć, czyli inaczej mówiąc sprawdzać kolejne wartości, aż do skutku.  
 $\text{stoję-przed-sklepem}_T \wedge \neg \text{pusta-torba}_T \wedge \neg \text{pełen-portfel}_T$

## Uwaga techniczna

Formuły zapisujemy często w bogatszym języku, zakładając, że system sam je przekształci do CNF-u.

- Warunki wstępne i końcowe poleceń

$$\text{strzelam}_t \rightarrow \text{mam-łuk}_t \wedge \text{mam-strzałę}_t \wedge \neg \text{mam-strzałę}_{t+1}$$

## Opisywanie akcji za pomocą formuł (2)

- Frame axioms (świat się za bardzo nie zmienia).

Przykład:

$$\text{smok-\'spi}_t \wedge \text{czytam-gazetę}_t \rightarrow \text{smok-\'spi}_{t+1}$$

(to dla wszystkich niewpływających na siebie par zmienne-stanu, akcja)

- Explanatory frame axioms

$$\text{smok-\'spi}_t \wedge \neg \text{smok-\'spi}_{t+1} \rightarrow (\text{rzucam-granat}_t \vee \text{gram-na-puzonie}_t \vee \dots)$$

- Informacja, że w każdym czasie wykonuję akcję (i tylko jedną – łatwa do zakodowania w CNF).

- Zasadniczo właśnie go opisaliśmy
- **Powtórka:** opisujemy świat, szukamy spełnialności formuły dla kolejnych  $T$ , jak znajdziemy, wypisujemy wartościowania, z którego odczytujemy sekwencję akcji.

Koniec części II

# Sposób definiowania logiki (ogólnie)

Musimy podać 3 składniki:

- ① **Składnię** – jak pisac formuły
- ② **Semantykę** – co znaczą formuły, kiedy są prawdziwe
- ③ **Reguły wnioskowania** – jak z prawdziwych formuł wnioskować inne, również prawdziwe

Logiki mają różną siłę wyrazu i dają procedury o różnej złożoności (musimy **balansować** pomiędzy siłą wyrazu a obliczeniową trudnością logiki)

## Uwaga

Reguły wnioskowania dotyczą syntaktyki, nie semantyki.

## Nasłyniejsza reguła wnioskowania

Reguła **modus ponens**: dla dowolnych zmiennych zdaniowych  $p$  i  $q$

$$\frac{p, p \rightarrow q}{q}$$

Można ją uogólnić do większej liczby przesłanek

$$\frac{p_1, \dots, p_n, p_1 \wedge \dots \wedge p_n \rightarrow p_{n+1}}{p_{n+1}}$$

Ogólna postać reguły wnioskowania jest następująca:

$$\frac{f_1, \dots, f_n}{g}$$

## Wnioskowanie w przód (forward inference)

Powtarzaj, aż do momentu, gdy nie da się zmienić Bazy wiedzy:

- Wybierz  $\{f_1, \dots, f_k\} \subseteq \mathcal{KB}$
- Jeżeli istnieje reguła:

$$\frac{f_1, \dots, f_n}{g}$$

dodaj  $g$  do Bazy wiedzy

## Definicja

Jeżeli powyższy algorytm dodaje  $f$  w którymś momencie do bazy wiedzy, wówczas piszemy  $\mathcal{KB} \vdash f$

## 2 proste uwagi o wnioskowaniu

- ① Wnioskowanie w tył: Zaczynamy od tego, co chcemy udowodnić (od naszego celu).
- ② Możemy myśleć o dowodzeniu twierdzeń jako o zadaniu przeszukiwania.  
(przestrzenią stanów są zbiory aksjomatów i dowiedzionych formuł, celem – zbiór zawierający docelowe twierdzenie )

## Przypomnienie

Formuła definiuje zbiór modeli  $\mathcal{M}$ , dla których jest ona prawdziwa. Podobnie można mówić o zbiorze modeli dla **bazy wiedzy** (czyli koniunkcji formuł).

## Definicja

Mówimy  $KB \models \phi$  wtedy i tylko wtedy, gdy każdy model KB będzie modelem  $\phi$  ( $\mathcal{M}(KB) \subseteq \mathcal{M}(\phi)$ ).

## Definicja 1

Logika jest **poprawna**, jeżeli  $M \vdash \phi$  implikuje  $M \models \phi$

## Definicja 2

Logika jest **zupełna**, jeżeli  $M \models \phi$  implikuje  $M \vdash \phi$

## Uwaga

Poprawność jest konieczna, zupełność – porządana.

- The truth, the whole truth, and nothing but the truth.

# Przykład. Zupełność (?) modus ponens

Modus ponens **nie** jest zupełny

## Przykład

$$\mathcal{KB} = \{\text{deszcz, deszcz} \vee \text{śnieg} \rightarrow \text{mokry}\}$$

**Mokry** jest prawdziwe, ale niedowodliwe.

# Sztuczna inteligencja. Logika, część II

Paweł Rychlikowski

Instytut Informatyki UWr

2 czerwca 2021

# Operacje na Bazie wiedzy

## Operacja **Tell(KB, $\phi$ )**

Dodaje formułę  $\phi$  do bazy wiedzy (proste dodanie do zbioru)

## Operacja **Ask(KB, $\phi$ )**

Sprawdza, czy  $KB \vdash \phi$  (czyli czy umiemy wyprowadzić  $\phi$  z KB).

Często realizujemy operację **Ask** sprawdzając, czy  $KB \wedge \neg\phi$  jest spełnialne/sprzeczne (dowód **nie wprost**).

## Pytanie

Czy można użyć tu algorytmu DPLL? A WalkSat?

# Operacje na Bazie wiedzy

## Operacja **Tell(KB, $\phi$ )**

Dodaje formułę  $\phi$  do bazy wiedzy (proste dodanie do zbioru)

## Operacja **Ask(KB, $\phi$ )**

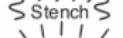
Sprawdza, czy  $KB \vdash \phi$  (czyli czy umiemy wyprowadzić  $\phi$  z KB).

Często realizujemy operację **Ask** sprawdzając, czy  $KB \wedge \neg\phi$  jest spełnialne/sprzeczne (dowód **nie wprost**).

## Pytanie

Czy można użyć tu algorytmu **DPLL**? A **WalkSat**?

# Modelowanie świata za pomocą logiki

4	 Stench		 Breeze	 PIT
3		 Breeze  Stench  Gold	 PIT	 Breeze
2	 Stench		 Breeze	
1	 START	 Breeze	 PIT	 Breeze
	1	2	3	4

- Wumpus śmierdzi, złoto błyszczy, w szybie są przeciągi.
- Poruszamy się o jedną kratkę w 4 kierunkach.
- Mamy jedną strzałę (strzela po liniach prostych).
- **Znamy mechanikę, ale nie znamy konkretnej edycji świata, odbieramy go za pomocą bodźców**

## Uwaga

Musimy opisać świat za pomocą skończonej liczby bitów

Przykładowe zmienne:

- ① Położenie dziur, wumpusa, złota:  $P_{1,2}, W_{4,4}, G_{3,2}$
- ② Położenie miejsc „z bodźcami”:  $S_{2,2}, B_{1,2}$
- ③ Położenie agenta:  $L_{3,3}^t$  (konieczne uwzględnienie czasu)
- ④ Wrażenia agenta:  $\text{Breeze}^t, \text{Stench}^t$
- ⑤ Stan agenta w chwili  $t$ , akcja agenta w chwili  $t$ , itd

# Przykładowe fragmenty modelu

- Jeżeli gdzieś jest przeciąg, to w okolicy jest dziura:

$$B_{1,1} \leftrightarrow P_{2,1} \vee P_{1,2}$$

- Jest (co najmniej) jeden Wumpus:  $W_{1,1} \vee W_{1,2} \vee \dots \vee W_{4,4}$

- Jest co najwyżej 1 Wumpus: (w każdych dwóch W co najmniej 1 fałszywy)

- Powiązanie wrażeń agenta ze światem:

$$L_{x,y}^t \rightarrow (\text{Breeze}^t \leftrightarrow B_{x,y})$$

## Uwaga

Potrzebujemy dla każdej akcji agenta opisać co się zmieni, a co nie zmieni w świecie.

Przykłady:

- $L_{1,1}^t \wedge \text{FacingEast}^t \wedge \text{Forward}^t \rightarrow (L_{2,1}^{t+1} \wedge \neg L_{1,1}^{t+1})$
- $\text{Forward}^t \rightarrow (\text{HaveArrow}^t \leftrightarrow \text{HaveArrow}^{t+1})$
- itd

Pamiętamy, że te reguły trzeba powtórzyć dla wszystkich lokacji (i dla różnych czasów, ale o tym za chwilę)

# Hybrydowy agent w świecie Wumpusa

- Wykorzystuje procedurę szukania drogi (poruszając się po polach bezpiecznych)
- Gromadzi wiedzę o świecie:
  - Zaobserwowane bodźce (i wnioski z nich płynące)
  - „Rozwijane” zdania o mechanice świata (dla momentu  $t$ )
- Zarządza **planem akcji**.

# Schemat agenta hybrydowego

Mamy listę akcji do zrobienia, czyli **Plan**

Dla momentu  $t$

- ① Rozejrzyj się (i dodaj do Bazy Wiedzy) formuły takie jak **Breeze $_t$** , **Stench $_t$** , ...
- ② Przeanalizuj bazę wiedzy, wyciągając możliwe konsekwencje
- ③ Czy trzeba zmieniać plan? Jeśli tak, to zmień (usuń wszystkie akcje, wstaw nowe)
  - **Uwaga:** musimy zmienić plan na przykład wówczas, jeżeli pierwsza akcja nie ma wypełnionych **wymagań wstępnych**
- ④ Dla akcji będącej pierwszą akcją planu wykonaj ją, czyli dodaj do bazy wiedzy formułę **Forward $_t$** , **Shoot $_t$** , ...

# Wumpus Agent (1)

```
function HYBRID-WUMPUS-AGENT(percept) returns an action
  inputs: percept, a list, [stench,breeze,glitter,bump,scream]
  persistent: KB, a knowledge base, initially the atemporal “wumpus physics”
             t, a counter, initially 0, indicating time
             plan, an action sequence, initially empty

  TELL(KB, MAKE-PERCEPT-SENTENCE(percept, t))
  TELL the KB the temporal “physics” sentences for time t
  safe  $\leftarrow \{[x, y] : \text{ASK}(KB, OK_{x,y}^t) = \text{true}\}$ 
  if ASK(KB, Glittert) = true then
    plan  $\leftarrow [Grab] + \text{PLAN-ROUTE}(\text{current}, \{[1,1]\}, \text{safe}) + [Climb]$ 
  if plan is empty then
    unvisited  $\leftarrow \{[x, y] : \text{ASK}(KB, L_{x,y}^{t'}) = \text{false} \text{ for all } t' \leq t\}$ 
    plan  $\leftarrow \text{PLAN-ROUTE}(\text{current}, \text{unvisited} \cap \text{safe}, \text{safe})$ 
```

# Wumpus Agent (2)

```
if plan is empty and ASK(KB, HaveArrowt) = true then
    possible_wumpus ← {[x, y] : ASK(KB, ¬ Wx,y) = false}
    plan ← PLAN-SHOT(current, possible_wumpus, safe)
if plan is empty then // no choice but to take a risk
    not_unsafe ← {[x, y] : ASK(KB, ¬ OKtx,y) = false}
    plan ← PLAN-ROUTE(current, unvisited ∩ not_unsafe, safe)
if plan is empty then
    plan ← PLAN-ROUTE(current, {[1, 1]}, safe) + [Climb]
action ← POP(plan)
TELL(KB, MAKE-ACTION-SENTENCE(action, t))
t ← t + 1
return action
```

# Jeszcze o logice zdaniowej

# Dygresja. Klauzule Hornowskie

## Definicja

Klauzula Hornowska to taka klauzula, która ma **co najwyżej** jeden literał pozytywny.

## Przykłady

- $p_1$  (fakty)
- $\neg p_2$  (zaprzeczenia faktów)
- $\neg p_2 \vee p_3$  (czyli  $p_2 \rightarrow p_3$ )
- $\neg q_1 \vee \dots \vee \neg q_n \vee q_{n+1}$  (czyli  $q_1 \wedge \dots \wedge q_n \rightarrow q_{n+1}$ )

## Uwaga

Klauzule Hornowskie mają duże znaczenie w Programowaniu logicznym (programy w Prologu składają się z klauzul hornowskich).

# Modus ponens i rezolucja

Regułę **modus ponens**: dla dowolnych zmiennych zdaniowych  $p$  i  $q$

$$\frac{p, p \rightarrow q}{q}$$

możemy zapisać tak:

$$\frac{p, \neg p \vee q}{q}$$

(**Intuicja**: skracanie  $p$  oraz  $\neg p$ )

Regułę powyższą można uogólnić tak, żeby operowała na dwóch dowolnych klauzulach, dających możliwość **skrócenia**.

## Uwaga

Rezolucję da się uogólnić tak, żeby działała dla **logiki pierwszego rzędu** (z kwantyfikatorami)

## Definicja

Reguła **Rezolucji** ma postać:

$$\frac{p_1 \vee \cdots \vee p_k \vee r, q_1 \vee \cdots \vee q_n \vee \neg r}{p_1 \vee \cdots \vee p_k \vee q_1 \vee \cdots \vee q_n}$$

- Działa na klauzulach
- Jest zupełna (z aksjomatami postaci  $a \vee \neg a \vee X$ )
- Proste ćwiczenie: pokaż, że  $a \vdash a \vee b$

# Koniec części I

Podstawowy brak: nie ma kwantyfikatorów, czyli pewne ogólne prawdy musimy wyrażać jako skończone alternatywy/koniunkcje.

## Przykłady

- Każdy student jest pilny
- Pilni studenci zdają egzaminy, na które są zapisani.
- Przynajmniej jedna osoba dostanie piątkę z AI

## Przykłady

- $\forall x \text{Student}(x) \rightarrow \text{Pilny}(x)$
- $\forall x \forall e \text{Student}(x) \wedge \text{Pilny}(x) \wedge \text{Zapisany}(s, e) \rightarrow \text{Zdaje}(s, e)$
- (...)

Jeżeli mówimy o skończonej liczbie obiektów, możemy traktować kwantyfikatory jako skróty dla koniunkcji ( $\forall$ ) lub alternatywy ( $\exists$ )

## Definicja

Logika, w której możemy używać kwantyfikatorów dla zmiennych pierwszego rzędu po „zwykłych” elementach.

**Przykłady** były na poprzednim slajdzie

## Twierdzenie 1

Logika pierwszego rzędu jest **nierzozstrzygalna** (aczkolwiek istnieją pewne rozstrzygalne fragmenty)

- Nie ma nadziei na program, który będzie umiał dowieść każdego twierdzenia logiki 1-go rzędu w skończonym czasie
- Istnieją wszakże programy dowodzące twierdzenia, bazujące na różnych heurystykach, na przykład **Otter**, **Vampire**, **Prover9**, ...

# Kilka faktów o logice pierwszego rzędu

## Definicja

**Fragment monadyczny** logiki pierwszego rzędu to taki podzbiór tej logiki, w którym nie mamy funkcji (choć możemy mieć stałe), ani symboli relacyjnych o arności większej niż 1.

## Przykład:

*Studenci chodzący na Sztuczną Inteligencję są niegłupi!*

$$\forall x(\text{Student}(x) \wedge \text{ChodziNaSI}(x) \rightarrow \text{NieGlupi}(x))$$

## Twierdzenie 2

**Fragment monadyczny** logiki pierwszego rzędu jest rozstrzygalny (spełnialność jest **NEXPTIME-zupełna**).

## Twierdzenie 3

Logika pierwszego rzędu z dwiema zmiennymi jest rozstrzygalna  
(spełnialność jest **NEXPTIME-zupełna**)

## Twierdzenie 4

Logika pierwszego rzędu z trzema zmiennymi jest nierozstrzygalna

## Uwaga

Różnych tego typu twierdzeń jest b. dużo. Różne formalizmy da się zredukować do logiki 1-go rzędu ograniczonej do jakiegoś konkretnego typu formuł.

# Czy komputery umieją dowodzić rzeczywiście ciekawe twierdzenia?

- W szczególności takie, z którymi ludzie mają kłopoty?
- (to nie jest oczywiste, choć można znaleźć przykłady, w dość specjalistycznych fragmentach matematyki)

Ale komputery potrafią sprawdzać dowody, asystować przy tworzeniu dowodów, sprawdzać przypadki, etc.

# O pomarańczach

Jaki jest związek poniższego obrazka z **Wielką Matematyką**



## Postulat Keplera (XVII w.)

Trójwymiarowe kule w trójwymiarowej przestrzeni najciśniej da się umieścić, gdy ich środki tworzą na płaszczyznach przekroju sześciokąty.

# O pomarańczach

Jaki jest związek poniższego obrazka z **Wielką Matematyką**



Twierdzenie Halesa (Thomas Hales, 2015)

Trójwymiarowe kule w trójwymiarowej przestrzeni najciśniej da się umieścić, gdy ich środki tworzą na płaszczyznach przekroju sześciokąty.

# O upakowaniu kul w przestrzeni

- Pierwsze doniesienia o dowodzie twierdzenia są z 1998
- Ogólna idea: **dowód na wyczerpanie** (możliwości lub czytelnika)
- Dowód rozpatruje tysiące przypadków i uzasadnia, że to są wszystkie alternatywy do rozpatrzenia.

# O upakowaniu kul w przestrzeni

Ostatecznie dowód został **przepisany** do języka logiki i **zweryfikowany** przez systemy wspomagające dowodzenie twierdzeń.

Podobna jest historia z twierdzeniem o 4 barwach (że każdą mapę da się pokolorować czterema kolorami (żeby żadne kraje o niezerowej wspólnej granicy nie miały tego samego koloru)):

- Dowód: 1976
- Formalna weryfikacja: 2004

## Definicja

Logiki modalne są rozwinięciami logiki zdaniowej o operatory modalności, które wyrażają na przykład:

- Właściwości czasowe (kiedyś, zawsze, jutro)
- Możliwość bądź konieczność czegoś
- Przekonanie lub wiedza agenta o czymś

Dla logiki temporalnej przyjmujemy często następujące aksjomaty (wybór):

- $\Box(\phi \rightarrow \psi) \rightarrow (\Box\phi \rightarrow \Box\psi)$  ( $\Box$  oznacza zawsze)
- $\Diamond\neg\phi \leftrightarrow \neg\Box\phi$  ( $\Diamond$  oznacza kiedyś)
- $\bigcirc(\phi \vee \psi) \leftrightarrow \bigcirc\phi \vee \bigcirc\psi$  ( $\bigcirc$  oznacza w kolejnym momencie czasu)

Dla każdego agenta  $a$  dodajemy modalność dotyczącą jego wiedzy, oznaczaną  $K_a$

Przykładowe akjomaty i ich interpretacja:

- Jak agent zna przesłanki i regułę, to zna też wnioski:

$$K_i\varphi \wedge K_i(\varphi \implies \psi) \implies K_i\psi$$

- Agenci znają tautologie

jeżeli  $M \models \varphi$  to  $M \models K_i\varphi$ .

- To co wiemy, jest prawdziwe

$$K_i\varphi \implies \varphi$$

# Wiem, że nic nie wiem

- Jak coś wiem, to wiem że to wiem

$$K_i\varphi \implies K_iK_i\varphi$$

- Jak czegoś nie wiem, to wiem że tego nie wiem

$$\neg K_i\varphi \implies K_i\neg K_i\varphi$$

# Zagadka z zabłoconymi dziećmi

(niestety jest mniej zabawna i trochę łatwiejsza, więc zamiast niej będzie)

# Zagadka z rogaczami

(z góry wszystkich przepraszam za pewne aspekty tej zagadki, z którymi mocno się nie zgadzam)

- ① Na wyspie mieszkają pary małżeńskie, wszyscy są mądrzy, logiczni i świadomi swojej mądrości.
- ② Niestety żony czasami zdradzają swoich mężów (mężowie pewnie też, ale zagadka o tym milczy).
- ③ Zdradzonemu mężowi wyrastają rogi. Wszyscy je widzą, nie mówi się o nich, mąż ich nie widzi.
- ④ Mężowie są strasznie honorowi: mąż, który dowie, że był zdradzony, zabija swoją żonę wieczorem, wrzuca ciało do rzeki i nad ranem inni znajdują zwłoki
- ⑤ Pewnego dnia na wyspę przyjechał Kuglarz, który zebrał całą ludność na placu i powiedział: są wśród was rogacze! Wszyscy popatrzyli bez słowa po sobie, rozeszli się. Po tygodniu wypłynęły zwłoki.

**Wyjaśnij, co się stało!**

Jak ktoś nie zna zagadki, to powinien przestać oglądać film teraz.  
Zwłaszcza, że na następnych slajdach w zasadzie nie będzie  
odpowiedzi

# Wspólna wiedza i najsłynniejsza zagadka logiki epistemicznej

## Uwaga

To że ja wiem **coś**, i ty wiesz, że ja wiem że **coś**, nie oznacza jeszcze, że ja wiem, że ty wiesz, że ja wiem **coś**.

- Wprowadza się specjalny operator **wiedzy powszechniej** (common knowledge)
- Definiujemy wiedzę grupową:

$$E_G \varphi \Leftrightarrow \bigwedge_{i \in G} K_i \varphi$$

- Wprowadzamy notację:

$$E_G^n \varphi \text{ definiujemy jako } E_G E_G^{n-1} \varphi$$

oraz  $E_G^0 \varphi = \varphi$

# Operator wspólnej wiedzy

- Definiujemy operator:

$$C_G \varphi \Leftrightarrow \bigwedge_{i=0}^{\infty} E_G^i \varphi$$

- Zdanie Kuglarza nie jest zdaniem o zerowej informacji: wprowadza ono bowiem do bazy wiedzy wszystkich agentów formułę:

$C_{\text{mieszkancywyspy}}$  ktoś-ma-rogi

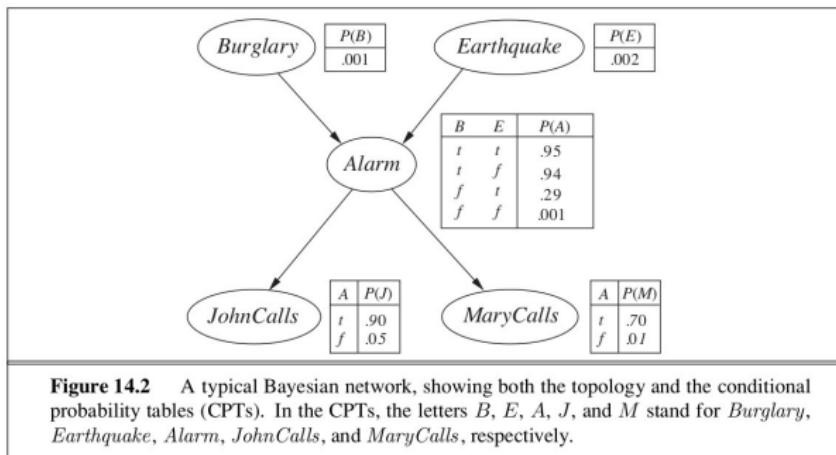
# Sztuczna inteligencja. Ostatni wykład o różnych rzeczach

Paweł Rychlikowski

Instytut Informatyki UWr

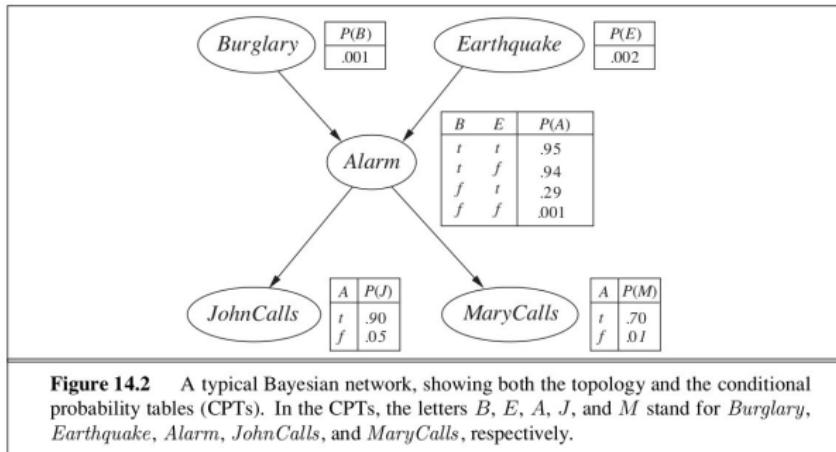
17 czerwca 2021

# Sieci Bayesowskie (1)



- Mamy alarm przeciw włamaniowy i dwoje sąsiadów Johna i Mary, którzy obiecali zadzwonić, jak alarm się włączy.
- Alarm uruchamia się również przy trzęsieniu ziemi (a mieszkamy na obszarze aktywnym sejsmicznie)
- **Problem:** Jakie jest prawdopodobieństwo włamania, jeżeli na przykład John zadzwonił, a Mary nie.

# Sieci Bayesowskie. Liczba parametrów



- Powyższą sieć opisuje **10** parametrów
- Mamy 5 zmiennych, ich łączny rozkład „standardowo” opisujemy za pomocą  $2^5 = 32$  parametrów (tak naprawdę 31)
- Zwięzlejszy opis łatwiej zapamiętać, łatwiej estymować parametry.

# Sieci Bayesowskie (3)

## Definicja

Niech  $(X_1, \dots, X_n)$  będą zmiennymi losowymi. **Siecią Bayesowską** (Bayesian network) nazwiemy DAG modelujący wspólny rozkład prawdopodobieństwa zmiennych  $X_i$  jako iloczyn lokalnych prawdopodobieństw warunkowych przypisanych do węzłów, określony wzorem:

$$P(X_1 = x_1, \dots, X_n = x_n) = \prod_{i=1}^n P(x_i | x_{\text{Parents}(i)})$$

(skrót notacyjny:  $P(X = x) = P(x)$ )

# Programy probabilistyczne

## Definicja

Probabilistyczny program to program, który symuluje sieć bayesowską. Można go traktować jako inny zapis sieci.

## Przykładowy program

```
# uwaga: trochę inny wariant niż na rysunku!
B = random.random() < p_burglary
E = random.random() < p_earthquake
A = B or E
if A:
    J = random.random() < pj_a
    M = random.random() < pm_a
else:
    J = random.random() < pj_not_a
    M = random.random() < pm_not_a
```

Zadajemy sieci pytania, na przykład: Jakie jest prawdopodobieństwo włamania, skoro dzwoni Mary, a nie dzwoni John?

Czyli pytamy o:  $P(B = 1|J = 1, M = 0)$ ? Jaki jest najprostszy (a zarazem uniwersalny) sposób na odpowiadanie na takie pytania?

## Uwaga

Uniwersalny sposób to **próbkowanie (sampling)**:

- 1 Generujemy dużo próbek (za pomocą zdefiniowanego przez sieć programu probabilistycznego)
- 2 Obliczamy stosunek (dla powyższego przykładu):

$$\frac{\text{liczba próbek z } (B, J, M) = (1, 1, 0)}{\text{liczba próbek z } (J, M) = (1, 0)}$$

## Definicja

*N-gramem* nazywamy ciąg kolejnych elementów o długości  $N$ .  
1-gramy to unigramy, 2-gramy to bigramy, 3-gramy to trigramy.

Za pomocą N-gramów tworzymy model języka, w którym staramy się przewidzieć kolejny element (o numerze  $N$ ) na podstawie  $N - 1$  elementów poprzednich.

Elementami mogą być litery, fonemy, słowa, sylaby, ...

# Prawdopodobieństwo sekwencji liter dla języka

Prawdopodobieństwo sekwencji liter można obliczyć następująco:

$$P(a_1 \dots a_n) = P(a_1)P(a_2|a_1)P(a_3|a_1a_2)\dots P(a_n|a_1 \dots a_{n-1})$$

(gdzie  $a_i$  to litery, spacja, interpunkcja)

„Dalsze” prawdopodobieństwa szacujemy patrząc nie na całą historię, lecz na  $N - 1$  liter poprzedzających znak przewidywany.  
Przykładowo dla  $N = 2$  mamy

$$P(a_1 \dots a_n) \approx P(a_1)P(a_2|a_1)P(a_3|a_2)P(a_4|a_3)\dots P(a_n|a_{n-1})$$

## Uwaga

Zauważmy, że dla zdania o długości  $K$  możemy stworzyć różne sieci Bayesowskie odpowiadające różnym sposobom modelowania tego zdania.

W takich sieciach zakładamy, że rozkłady w poszczególnych węzłach są takie same!

Przykładowo dla bigramów mamy:

- $P(a_2|a_1) = \frac{P(a_1a_2)}{P(a_1)}.$
- $P(a_1a_2) = \frac{\text{cnt}(a_1a_2)}{N-1}.$
- $P(a_1) = \frac{\text{cnt}(a_1)}{N}.$

Oczywiście przyjmujemy, że  $N \approx N - 1$ , co upraszcza wzory.

# Generowanie pseudo tekstów w danym języku

## Generator 3-gramowy

- Dla każdej pary znaków pamiętamy, jakich ma możliwych następców (i z jakim prawdopodobieństwem).
- Zaczynamy od wylosowania pary znaków (z tych, które mają następnika)
- Dla pary  $a_1a_2$  losujemy następnika ( $a_3$ )
- Czynności powtarzamy dla pary  $a_2a_3$ .

## Uwaga

Taki generator jest **programem probabilistycznym** dla sieci Bayesowskiej opisującej zdanie (w wersji 3-gramowej).

- Zobaczmy, czy generowane w ten sposób teksty przypominają języki, z których czerpaliśmy rozkład.
- Tekst dla języka polskiego wyglądać może tak (model 4 gramowy):  
*-Curtki wiedzientów, Chińczy mał wzdługo skrętać w do Katorby z maszycję tań niemczajęło wscha pociałem okole-głosy, że mój przygodziwielsku. Zaczastepnego nigdy sto-nie mał mechała to się z naliśmy na kontru to nienia się zbyt dużej przebiję, że to zakładna eneś za pojego drzemu zbie zdołamięszczoraj ta*

# Zgadywanie (1)

-Antout demangue volla fois vous devent-là de la fait de fait viller ce inte que  
je vra lors magis, vournie donne il travec que disponis trop de réussions sonné  
au de toutera-t-il metitiques, il faire semblement d'adrontrique trop suivière u  
ne né à consi dimauvaissé, heur nominus ce pours seul dit en ve

## Odpowiedź

Tekst utworzony za pomocą modelu francuskiego

# Zgadywanie (2)

Sydneš bývala v se, nejsem němí před se pracuji, každý musím třídit to, co mít schodili jsem Green už všemusedla cigarelefon ve nám vzal jsme šokonců bezce pořádkakupile je v té vůbecky se zná učím neko pro noc mléčné poda nemoc byla sobil a a to zábal názor, že se hlavír zpívala dneměl byl pokaždě tro

## Odpowiedź

Tekst utworzony za pomocą modelu czeskiego

# Zgadywanie (3)

Mae Warstift und andet unbedem keinigte Spaß gest ohnhof Mauert „sheraden letzte  
soll der das ein gehen Englief andessantischen spielerangenu, niemache stücks  
auf sich einflangs ist das Tom und ich nich sein ihereich ihm, das Dorf ich bitte  
n einsamtester sich wieder Aufgehört mich ein reppt; werdige is

## Odpowiedź

Tekst utworzony za pomocą modelu niemieckiego

# Zgadywanie (4)

e, pero adiervar por factimadre el a ahora mieda que salumera de repo portunión  
mi me lejos pluminalejorese de mor mundos y lógica ir a mucho, y es qué hos ante  
renzó a un jabólicinal y des en contos, hizo las tencié mirarán saberías, usted  
es los selversonito doro país en aquí puedaré juegos oír dema d

## Odpowiedź

Tekst utworzony za pomocą modelu hiszpańskiego.

# Zadanie

Jakie jest prawdopodobieństwo, że w polskim zdaniu wystąpi wzorzec:

a \* \* a \* \* a \* \* a

gdzie \* oznacza dowolny znak?

Dwie strategie rozwiązania (założymy, że mamy reprezentatywną próbkę miliona polskich zdań):

- a) Policzyć zdania ze wzorcem, podzielić przez milion
- b) Stworzyć model generujący polskie zdania, wygenerować 100 milionów przykładów, policzyć wzorce.

Drugi wariant jest czasem lepszy (bo na przykład prawdopodobieństwo jest mniejsze niż  $\frac{1}{1000000}$ )

- Generowanie muzyki (była jakaś nutka, albo kilka, jakie jest prawdopodobieństwo kolejnej)
- Opisywanie tekstu (czym jest konkretne wystąpienie słowa)
- Rozpoznawanie mowy (do niedawna absolutny lider, ciągle użyteczny)

# Spacjowanie Pana Tadeusza

- Pamiętamy zadanie z pierwszej listy o wstawianiu spacji w sklejonym tekście.
- W rzeczywistości rozwiązywalibyśmy je trochę inaczej: korzystając z wiedzy o języku
- Wiedzę taką możemy nabyć analizując duże zbiory tekstów, tzw. **korporusy**.

## Definicja

Model języka określa prawdopodobieństwo tego, że dany ciąg (słów, liter) jest poprawnym napisem języka.

Pozwala wybierać pomiędzy różnymi wariantami wypowiedzi (poprawy literówek, OCR, rozpoznawanie mowy, tłumaczenie)

- Najprostszy model języka to model **unigramowy**: rozważamy prawdopodobieństwo wystąpienia słów.
- Żeby zdecydować jak spacjować **partiachciała** sprawdzamy czy:

$$P(\text{partiach})P(\text{ciała}) > P(\text{partia})P(\text{chciała})$$

- Szacujemy:

$$P(\text{partiach}) \approx \frac{\text{ile razy słowo „partiach” było w korpusie}}{\text{liczba słów w korpusie}}$$

# Problemy modelu unigramowego

- Preferuje długie wyrazy (no i ok?) – ew. można temu zaradzić biorąc średnią geometryczną prawdopodobieństwa
- Nie uwzględnia następst wyrazów.

## Definicja

W **modelu bigramowym** zakładamy, że:

$$P(w_1 \dots w_n) = P(w_1 | [start]) P(w_2 | w_1) \dots P(w_n | w_{n-1})$$

# Algorytm Viterbiego

- Dynamiczny algorytm, wybierający sekwencję maksymalizującą prawdopodobieństwo w modelu bigramowym nazywa się **Algorytmem Viterbiego**
- Bardzo podobny do tego, jak rozwiązywaliśmy zadanie ze spacjowaniem.
- Drobna różnica:
  - a) w zadaniu z listy, wystarczyło pamiętać, jaki jest koszt (i kształt) optymalnej ścieżki kończącej się na danej literce.
  - b) tu musimy pamiętać te dane dla każdej literki i **każdego wyrazu, który może skończyć się w tym miejscu**

Koniec części I

- Zamiast tworzyć skomplikowanego agenta, grającego w **trudną grę**
- możemy stworzyć prostego agenta, grającego w **łatwą grę**

## Przykład: aukcja

Rozważamy prostą, jednoturową aukcję: oferenci piszą swoje ceny, wygrywa największa, przedmiot sprzedajemy za tę cenę

Co jest nie tak z tą aukcją?

## Problemy

Co powinien robić agent, który jest w stanie kupić przedmiot za  $x$ ?

- Złożyć ofertę za  $x$ ? (jak przegra, to trudno – nie dało się nic zrobić, ale jak wygra, to może przepłaci)
- Złożyć ofertę za  $y < x$  (ale jakie  $y$ ? Musi modelować innych graczy i być lepszy o  $\varepsilon$  od najlepszego z nich)

A jak działałaby aukcja, w której zwycięzca płaciłby cenę drugą z kolei?

# Aukcja: wygrywający płaci drugą cenę

Co jest optymalną strategią gracza dla aukcji Vickreya?

Optymalną strategią jest wypisać swoją cenę i nie przejmować się innymi graczami, bo:

- Jak napiszę za dużo, to być może okażę się niewypłacalny (duża przegrana)
- Nie mam też żadnego interesu w zaniżaniu swojej stawki
  - Wpiszę mniej i wygram – i tak płacę tę samą cenę
  - Wpiszę mniej i przegram – ale może dało się wygrać!

## Zadanie

Stworzyć program, który będzie odkrywał ciekawe gry planszowe.

Dwa problemy do rozwiązania:

1. Jak opisać grę planszową?
2. Jak odróżnić fajną grę planszową od słabej? (ciekawsze pytanie)

## Co powinna uwzględniać funkcja oceny?

### Kryteria używane w ocenie

- **Wewnętrzne** – jak wygląda opis gry (preferujemy poprawne, niezbyt długie i niezbyt skomplikowane)
- **Grywalność** – zbalansowana dla obu graczy, niezbyt dużo remisów, gry o satysfakcjonującej długości
- **Jakość rozgrywki:**
  - Dramatyzm: wyuczona funkcja oceniająca zmienia wartość w trakcie prawdziwych gier (najlepiej, żeby zmieniała się również przewaga)
  - Nieredukowalność: w rozegranych grach używane są wszystkie rodzaje ruchów
  - Krzywa uczenia: agent uczący się dłużej gra lepiej.

Oczywiście ważnym narzędziem jest TD-learning (żeby powstały sensowni agenci, których grę możemy analizować).



# Jak zarobić na ewolucji gier planszowych

- System **Ludi** służył do ewolucji gier planszowych.
- Podczas tygodnia ewolucji przeanalizowano **1389**, z czego **19** autorzy uznali za grywalne, a dwie, jak piszą **have proven to be of exceptional quality**
- Zapakowali je do pudełka i sprzedawali jako **pierwsze gry planszowe wymyślone przez maszynę**.

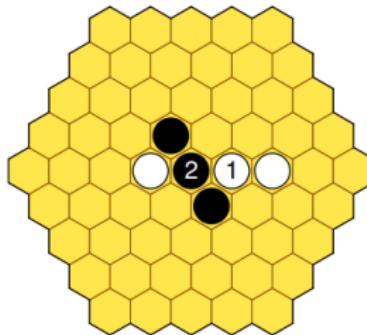
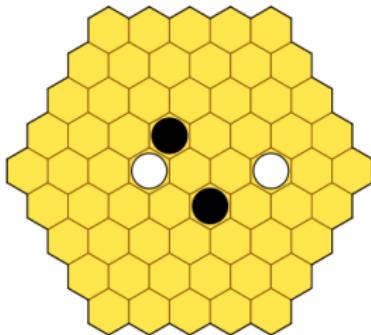
Można poczytać w [pcgbook.com](http://pcgbook.com), rozdział 6

# Gra Yavalath

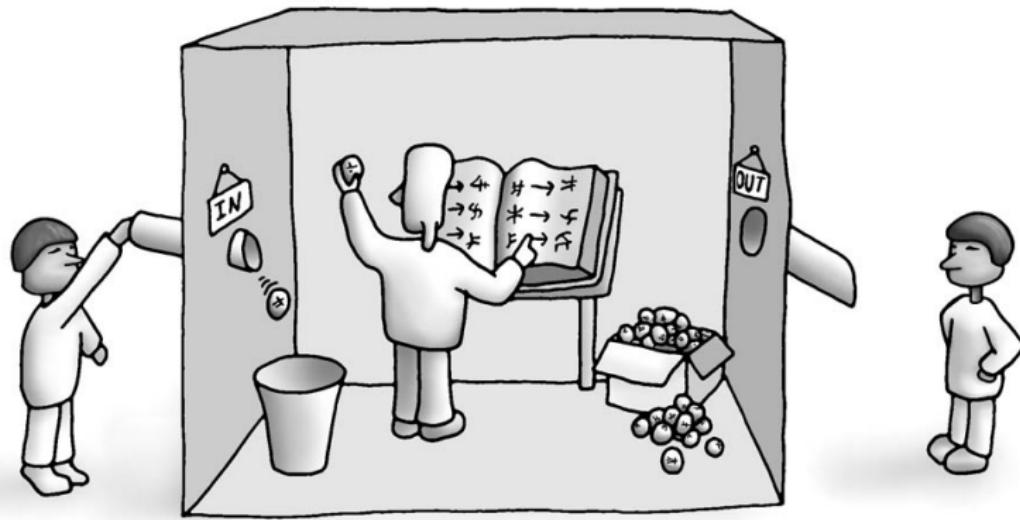
## Kod gry

```
(game Yavalath
  (players White Black)
  (board (tiling hex) (shape hex) (size 5))
  (end (All win (in-a-row 4)) (All lose (in-a-row 3)))
)
```

Przykładowa sytuacja:



# Chiński pokój



jolyon.co.uk

## Uwaga

Na razie tego wykładu nie ma, ale ...

gdyby był, to

- ① wiele na nim mówiłoby się o **uczeniu ze wzmocnieniem**
- ② oraz o **dowodzeniu twierdzeń**

Ostatnio te zagadnienie się ze sobą łączą

Przykładowe publikacje

- **DeepHOL, Learning to Reason in Large Theories without Imitation**, Kshitij Bansal Christian Szegedy Markus N. Rabe Sarah M. Loos Viktor Toman
- **Reinforcement Learning of Theorem Proving**, Cezary Kaliszyk, Josef Urban, Henryk Michalewski, Mirek Olsak

Idea: MCTS nie w grach dla zabawy, lecz w „poważnych grach z jednym graczem”



- Komputer wygrywa Międzynarodową Olimpiadę Matematyczną! (kiedyś)
- Formalizacja zadań w języku **Lean**
- Do tego języka tłumaczona jest cała matematyka szkolna (i być może „licencjacka”) (w Imperial College)