

ALGORYTM - przypomnienie podstawowych pojęć

- **Algorytmika** jest dziedziną wiedzy zajmującą się badaniem **algorytmów**;
- W informatyce jest ona nieodłącznie związana z algorytmami przetwarzania **struktur danych**;
- Potocznie **algorytm** jest rozumiany jako pewien przepis na wykonanie jakiegoś zestawu czynności, prowadzących do osiągnięcia oczekiwanej i z góry określonego celu;
- Mówi się również, że:
 - **algorytm** jest pewną ściśle określoną procedurą obliczeniową, która dla zestawu właściwych danych wejściowych wytwarza żądane dane wyjściowe;
 - **algorytm** jest to zbiór reguł postępowania umożliwiających rozwiązanie określonego zadania w skończonej liczbie kroków i w skończonym czasie.

Termin **algorytm** wywodzi się od zlatynizowanej formy (*Algorismus, Algorithmus*) nazwiska matematyka arabskiego z IX w., **Al-Chuwarizmiego**.

ALGORYTM - przypomnienie podstawowych pojęć

W sposób formalny algorytm możemy zdefiniować następująco:

Oznaczmy przez:

We - zestaw danych wejściowych,
Wy - zestaw danych wyjściowych.

Algorytm jest rozumiany jako odwzorowanie **O**, które dla określonego zestawu **We** generuje zestaw **Wy**:

O: We → Wy,

gdzie liczności zbiorów **We** i **Wy** mogą być różne.

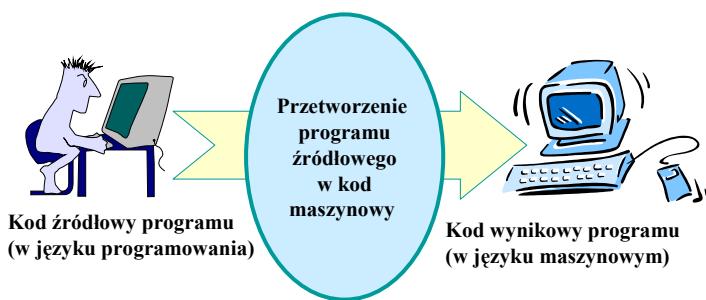
Sposoby zapisu algorytmów

- Algorytm powinien *precyzyjnie* przedstawiać kolejne jego kroki. Do opisu tych kroków mogą być stosowane następujące sposoby:
 - ◆ zapisy werbalne,
 - ◆ zapisy formalne, np.:
 - zapisy graficzne (schematy blokowe),
 - formalne specyfikacje programów (VDM, CSP)
 - zapisy w postaci pseudokodów („paraprogramów”)
 - implementacje programów w dowolnym *języku programowania*



ALGORYTM - przypomnienie podstawowych pojęć

- **Język programowania** jest środkiem umożliwiającym zapis algorytmów w postaci zrozumiałej dla człowieka, a równocześnie przetwarzanej do postaci zrozumiałej dla komputera (**maszyny algorytmicznej**);



Klasyfikacja algorytmów (wybrane kategorie)

- **algorytmy proste – rozgałęzione** (nie występuje albo występują rozgałęzienia),
- **algorytmy cykliczne – mieszane** (z powrotami albo bez powrotów),
- **algorytmy równolegle - sekwencyjne** (kroki algorytmu wykonywane kolejno w sekwencji lub równolegle),
- **algorytmy numeryczne - algorytmy nienumeryczne** (wykonywanie obliczeń lub przetwarzanie danych),
- **algorytmy rekurencyjne - algorytmy iteracyjne** (algorytm w kolejnych krokach wywołuje sam siebie dla nowych wartości parametrów wykonania lub wykonuje obliczenia w pętli dla zmieniającej się wartości jej niezmiennika).

Klasyfikacja algorytmów (wybrane kategorie), c.d.

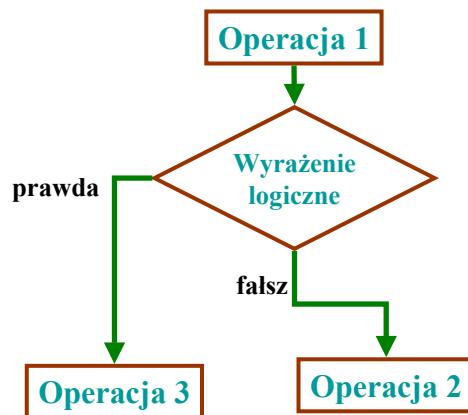
- **algorytmy zachłanne** (wykonują działanie które wydaje się najlepsze w danej chwili, nie uwzględniając tego co może się stać w przyszłości);
- **algorytmy „dziel i zwyciężaj”** (dzielimy problem na mniejsze części tej samej postaci co pierwotny , aż rozmiar problemu stanie się tak mały, że rozwiązanie będzie oczywiste lub będzie można użyć jakiejś innej efektywnej metody rozwiązania);
- **algorytmy oparte o programowanie dynamiczne** (wykorzystują własność niektórych problemów polegającą na tym, że optymalne rozwiązanie w iteracjach wcześniejszych gwarantuje otrzymanie optymalnego rozwiązania w kolejnych iteracjach);
- **algorytmy z powrotami** (wymagają zapamiętania wszystkich wykonanych ruchów czy też wszystkich odwiedzonych stanów aby możliwe było cofanie posunięć w celu dojścia do rozwiązania);

Algorytm prosty, a rozgałęziony

Proste



Rozgałęzione

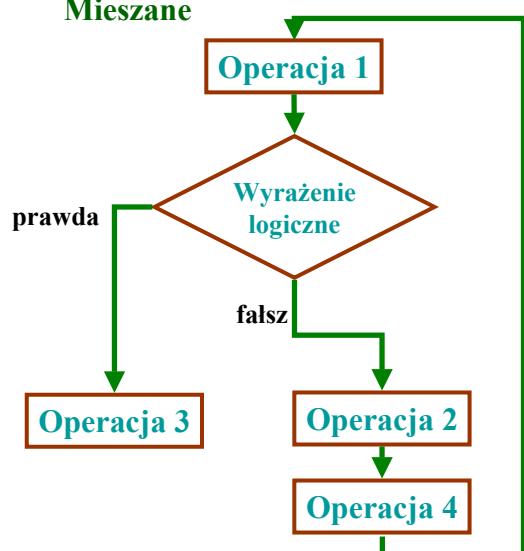


Algorytm cykliczny, a mieszany

Cykliczne

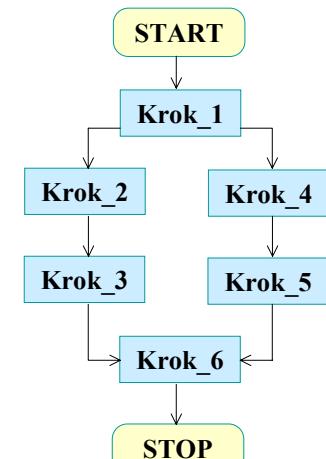


Mieszane

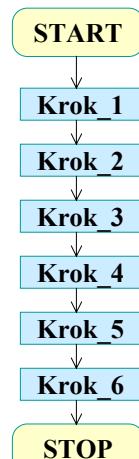


Algorytm równoległy, a sekwencyjny

Algorytm równoległy:



Algorytm sekwencyjny:

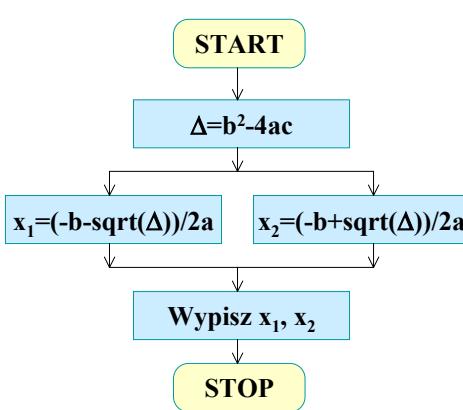


Z.Tarapata, Algorytmy i struktury danych, wykład nr 1

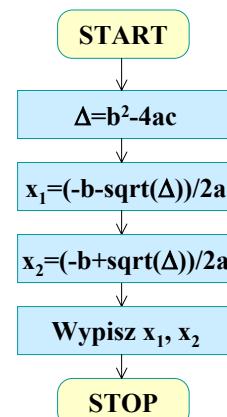
11

Algorytm równoległy, a sekwencyjny

Algorytm równoległy:



Algorytm sekwencyjny:

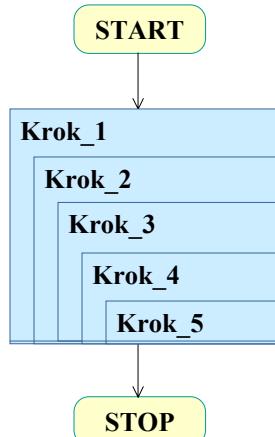


Z.Tarapata, Algorytmy i struktury danych, wykład nr 1

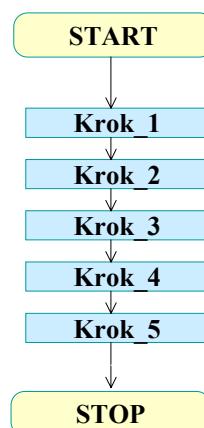
12

Algorytm iteracyjny, a rekurencyjny

Algorytm rekurencyjny:



Algorytm iteracyjny:



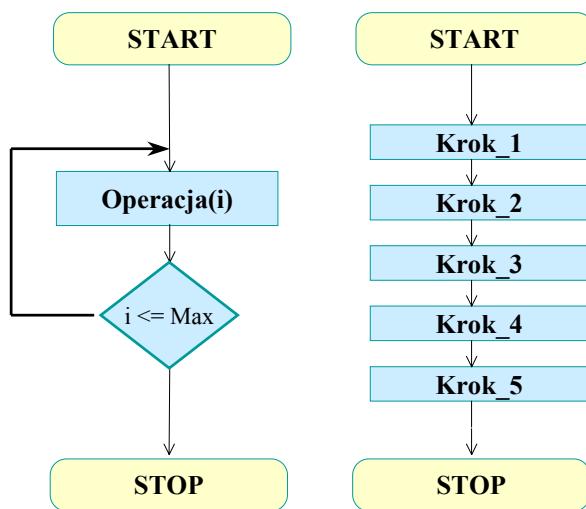
Z.Tarapata, Algorytmy i struktury danych, wykład nr 1

13

Algorytmy iteracyjne

Przykład:

```
silnia=1;  
for(i=2;i<=5;++i)  
    silnia = silnia * i;  
return silnia;
```



Z.Tarapata, Algorytmy i struktury danych, wykład nr 1

14

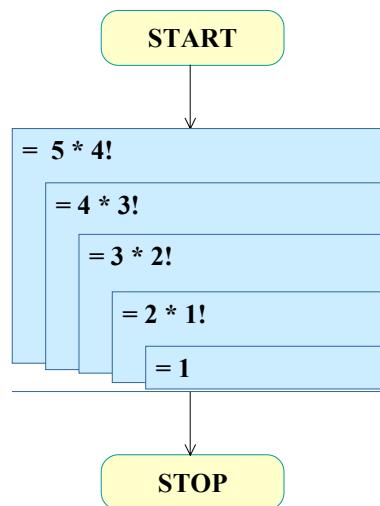
Algorytmy rekurencyjne

Algorytm wyliczający silnię.

Silnia n!:

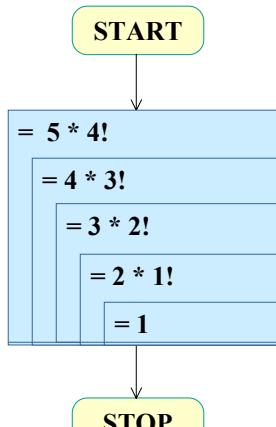
Jeśli $n=0$ lub $n=1$, to $n!=1$

Jeśli $n > 1$, to $n! = n * (n-1)!$

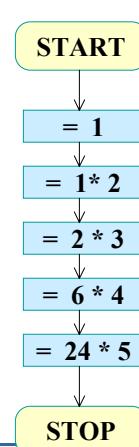


Algorytmy iteracyjne i rekurencyjne - porównanie

Algorytm rekurencyjny:



Algorytm iteracyjny:

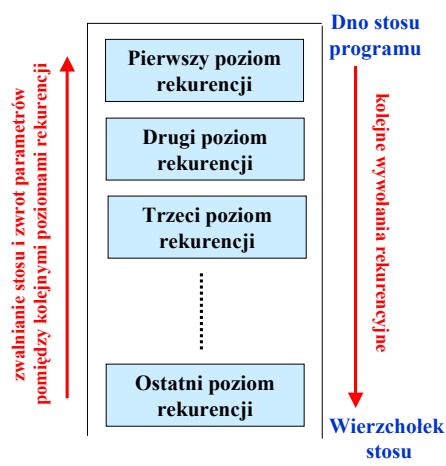


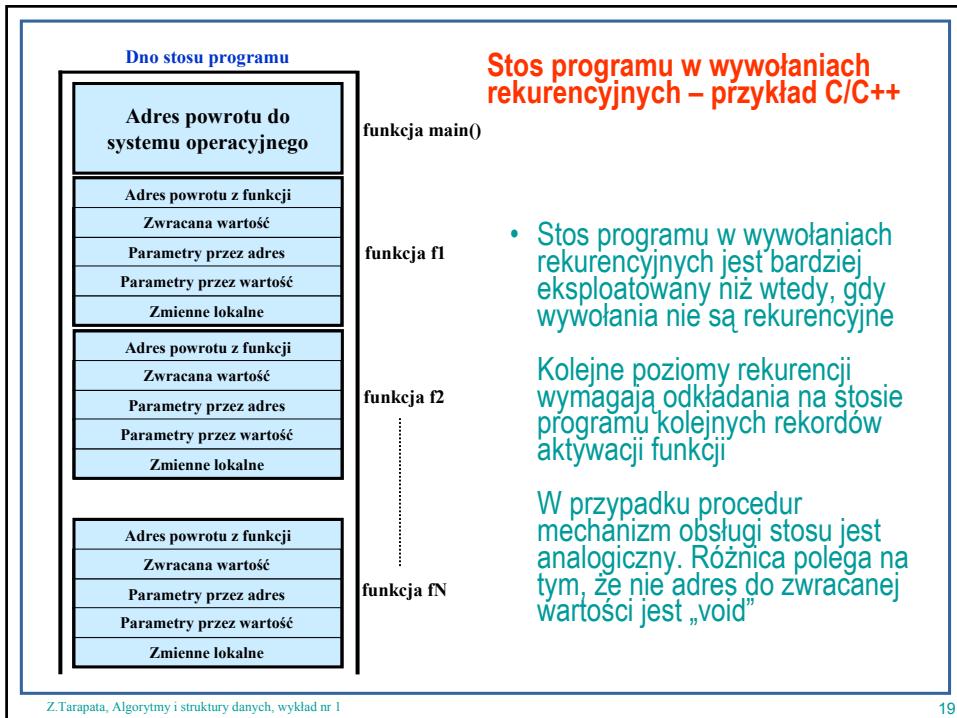
Wywołanie funkcji rekurencyjnej

- **Rekurencja oznacza wywołanie funkcji (procedury) przez tę samą funkcję**
- **Ważne jest, aby kolejne wywołania funkcji (procedury) rekurencyjnej były realizowane dla kolejnych wartości parametrów formalnych w taki sposób, aby nie doszło do zjawiska „nieskończonej pętli rekurencyjnych wywołań funkcji”**

Wywołanie funkcji rekurencyjnej

- Kolejne wywołania funkcji rekurencyjnej są związane z odkładaniem na stosie programu kolejnych rekordów aktywacji procedury
- W wyniku kończenia działania poszczególnych funkcji na kolejnych poziomach rekurencji kolejne rekordy aktywacji są zdejmowane ze stosu





Przykłady funkcji rekurencyjnych

- Znana już funkcja „silnia”:

$$n! = \begin{cases} 1, & \text{dla } n=0 \text{ (warunek zakończenia rekurencji)} \\ n * (n-1)! & \text{dla } n>0 \end{cases}$$

Definicja ciągu liczb wymiernych:

$$f(n) = \begin{cases} 1, & \text{dla } n=0 \text{ (warunek brzegowy, zakończenia)} \\ f(n-1) + (1/f(n-1)), & \text{dla } n>0, \text{ określa ciąg o wartościach:} \end{cases}$$

1, 2, 5/2, 29/10, 941/290, 969581/272890,.....

Funkcja rekurencyjna – ciąg liczb Fibonacciego

- Ciąg liczb Fibonacciego jest wyliczany wg formuły:

$$\text{Fib}(n) = \begin{cases} n, & \text{dla } n < 2 \\ \text{Fib}(n-2) + \text{Fib}(n-1), & \text{dla } n \geq 2 \end{cases}$$

- Rekurencyjna implementacja w języku C:

```
long intFib (int n)
{
    if (n<2)
        return n;
    else
        return Fib(n-2) + Fib (n-1);
}
```

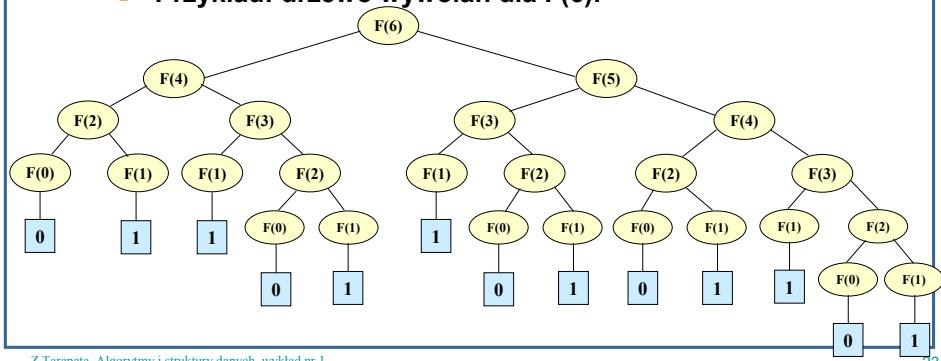
Czy na pewno stos programu „wytrzyma” taką realizację funkcji rekurencyjnej Fib?

Efektywność rekurencyjnego wykonania funkcji Fibonacciego

n	Fib(n+1)	Liczba dodawań	Liczba wywołań
6	13	12	25
10	89	88	177
15	987	986	1 973
20	10 946	10 945	21 891
25	121 393	121 392	242 785
30	1 346 269	1 346 268	2 692 537

Efektywność rekurencyjnego wykonania funkcji Fibonacciego, cd.

- Okazuje się więc, że rekurencyjna implementacja funkcji Fibonacciego jest niezwykle nieefektywna. Stos programu nie jest praktycznie w stanie zrealizować tego algorytmu już dla liczb większych od 9. Oznacza to, że program ma zbyt dużą „złożoność pamięciową”.
- Przykład: drzewo wywołań dla $F(6)$:



Z.Tarapata, Algorytmy i struktury danych, wykład nr 1

23

Iteracyjne wykonanie rekurencyjnej funkcji Fibonacciego

- Bardziej efektywna jest iteracyjna implementacja funkcji Fibonacciego. Nie przepełniamy wtedy stosu programu i wykonujemy mniejszą liczbę przypisań wartości niż w implementacji rekurencyjnej, dla której liczba dodawań wynosi $\text{Fib}(n+1)-1$, natomiast liczba przypisań jest równa: $2 \cdot \text{Fib}(n+1)-1$.
- Przykład implementacji funkcji Fibonacciego metodą iteracyjną:

```
long int IteracyjnyFib(int n)
{ register int i=2, last=0, tmp; long int current =1;
  if (n<2)
    return n;
  else {
    for ( ; i<=n; ++i) {
      tmp = current;
      current += last;
      last = tmp;
    }
    return current;
  }
}
```

Z.Tarapata, Algorytmy i struktury danych, wykład nr 1

24

Efektywność iteracyjnego wykonania rekurencyjnej funkcji Fibonacciego

n	Liczba przypisań dla algorytmu iteracyjnego	Liczba przypisań (wywołań) dla algorytmu rekurencyjnego
6	15	25
10	27	177
15	42	1 973
20	57	21 891
25	72	242 785
30	87	2 692 537

Algorytmy zachłanne

Wykonują działanie które wydaje się najlepsze w danej chwili, nie uwzględniając tego co może się stać w przyszłości. Zaletą jest to że nie traci czasu na rozważanie co może się stać później. Zadziwiające jest że w wielu sytuacjach daje on optymalne rozwiązanie.

Technicznie mówimy: decyzja lokalnie optymalna.

Jak wydać resztę przy minimalnej ilości monet?:

użyj zawsze najpierw monetę o największej dopuszczalnej wartości.

Jak znaleźć globalne maximum? Rozpocznij od pewnej liczby, kolejno powiększaj ją o ustaloną wielkość tak długo jak funkcja wzrasta. Gdy wartość funkcji zaczyna się zmniejszać przerwij i cofnij się do ostatniej pozycji.

Algorytmy „dziel i zwyciężaj”

- Dzielimy problem na mniejsze części tej samej postaci co pierwotny.
- Teraz te podproblemy dzielimy dalej na coraz mniejsze, używając tej samej metody, aż rozmiar problemu stanie się tak mały, że rozwiązanie będzie oczywiste lub będzie można użyć jakiejś innej efektywnej metody rozwiązania.
- Rozwiązania wszystkich podproblemów muszą być połączone w celu utworzenia rozwiązania całego problemu.

Metoda zazwyczaj implementowana z zastosowaniem technik rekurencyjnych.

Jak znaleźć minimum ciągu liczb?: Dzielimy ciąg na dwie części, znajdujemy minimum w każdej z nich, bierzemy minimum z obu liczb jako minimum ciągu.

Jak sortować ciąg liczb?: Dzielimy na dwie części, każdą osobno sortujemy a następnie łączymy dwa uporządkowane ciągi (scalamy).

Właściwości algorytmów

- **adekwatność** - czy algorytm realizuje obliczenia (przetwarzanie) zgodnie z przyjętym celem realizacji obliczeń (przetwarzania)
- **właściwość stopu** - zostały zdefiniowane kryteria zatrzymania wykonywania algorytmu w warunkach poprawnego i niepoprawnego zakończenia obliczeń
- **jednoznaczność** - algorytm jest zapisany w sposób na tyle precyzyjny, że jego wykonanie jest prawie automatycznym powtarzaniem kolejnych kroków
- **powtarzalność** - każde wykonanie algorytmu przebiega według takiego samego schematu działania i prowadzi do tej samej klasy rozwiązań
- **złożoność obliczeniowa** - nakład czasu lub zasobów maszyny realizującej algorytm, niezbędny dla jego prawidłowego wykonania

Złożoność obliczeniowa algorytmów - analiza algorytmów

- **Analiza algorytmów** - złożoność obliczeniowa jest podstawową własnością określana dla algorytmów. Zadaniem analizy algorytmu jest określenie tej złożoności, a co za tym idzie *realizowalności algorytmu*.
- **Złożoność zasobowa (pamięciowa)** - wyrażana w skali zajętości zasobów (PAO, pamięci zewnętrznych itp.) niezbędnych dla realizacji algorytmu
- **Złożoność czasowa** - wyrażana w skali czasu wykonania algorytmu (liczba kroków, aproksymowany czas rzeczywisty)

SZACOWANIE ZŁOŻONOŚCI OBliczeniowej

Oznaczenia:

α - algorytm rozwiązujący decyzyjny problem Π ;
 D_n - zbiór danych rozmiaru n dla rozważanego problemu;
 $t(I)$ - liczba kroków DTM potrzebna do rozwiązania konkretnego problemu $I \in D_n$ o rozmiarze $N(z) = n$ przy pomocy algorytmu α .

Definicja

Złożonością obliczeniową (pesymistyczną) algorytmu α nazywamy funkcję postaci

$$(1) \quad W_\alpha(n) = \max \{ t(I) : I \in D_n \}$$

SZACOWANIE ZŁOŻONOŚCI OBliczeniowej

Definicja

Mówimy, że algorytm α ma złożoność obliczeniową wielomianową, jeśli istnieje stała $c > 0$ oraz wielomian $p(n)$ takie, że:

$$\bigwedge_{n \geq n_0} W_\alpha(n) \leq cp(n)$$

co zapisujemy $W_\alpha(n) = O(p(n))$.

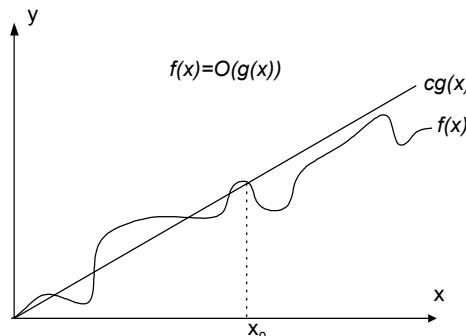
W innych przypadkach mówimy, że algorytm α ma złożoność wykładniczą.

SZACOWANIE ZŁOŻONOŚCI OBliczeniowej

Definicja rzędów złożoności obliczeniowej

Niech $R^* = R^+ \cup \{0\}$.

Mówimy, że funkcja $f(x):R^* \rightarrow R^*$ jest rzędu¹ $O(g(x))$ ($g(x):R^* \rightarrow R^*$), jeśli istnieje taka stała $c > 0$ oraz $x_0 \in R^*$, że dla każdego $x \geq x_0$ zachodzi $f(x) \leq c \cdot g(x)$ (f nie rośnie szybciej niż g).

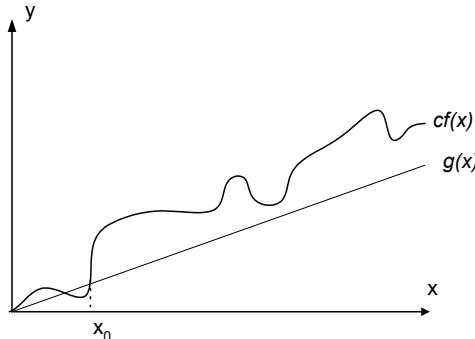


[1] $f(x) = O(g(x))$, gdy $\lim_{x \rightarrow \infty} \frac{f(x)}{g(x)} = c$ dla pewnego $c \geq 0$.

SZACOWANIE ZŁOŻONOŚCI OBliczeniowej

Definicja rzędów złożoności obliczeniowej, c.d.

Mówimy, że funkcja $f(x):R^* \rightarrow R^*$ jest rzędu² $\Omega(g(x))$ ($g(x):R^* \rightarrow R^*$), jeśli istnieje taka stała $c > 0$ oraz $x_0 \in R^*$, że dla każdego $x \geq x_0$ zachodzi $g(x) \leq c f(x)$ (f nie rośnie wolniej niż g).

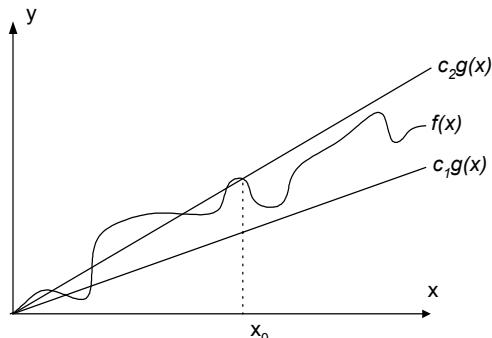


[2] $f(x) = \Omega(g(x)), \text{ gdy } \lim_{x \rightarrow \infty} \frac{f(x)}{g(x)} = \infty \text{ lub } \lim_{x \rightarrow \infty} \frac{f(x)}{g(x)} = c > 0.$

SZACOWANIE ZŁOŻONOŚCI OBliczeniowej

Definicja rzędów złożoności obliczeniowej, c.d.

Mówimy, że funkcja $f(x):R^* \rightarrow R^*$ jest rzędu $\Theta(g(x))$ ($g(x):R^* \rightarrow R^*$), jeśli istnieją takie stałe $c_1, c_2 > 0$ oraz $x_0 \in R^*$, że dla każdego $x \geq x_0$ zachodzi $c_1 g(x) \leq f(x) \leq c_2 g(x)$ (f rośnie tak samo jak g).



Jeżeli funkcja $f(x)$ jest $O(g(x))$ oraz $\Omega(g(x))$, to jest $\Theta(g(x))$.

SZACOWANIE ZŁOŻONOŚCI OBliczeniowej

Definicja rzędów złożoności obliczeniowej, Przykład

Przykład 1

- ◆ $2 |\sin x| = O(\log x), \quad 2 |\sin x| = O(1)$
- ◆ $x^3 + 5x^2 + 2 = O(x^3)$ oraz $x^3 + 5x^2 + 2 = \Omega(x^3)$, więc
 $x^3 + 5x^2 + 2 = \Theta(x^3)$
- ◆ $f(x) = x^2 + bx + c = \Theta(x^2)$,
gdyż $\forall x \geq x_0 = 2 \cdot \max\left\{\frac{|b|}{a}, \sqrt{\frac{|c|}{a}}\right\}$ zachodzi:
 $c_1 g(x) \leq f(x) \leq c_2 g(x)$ dla $c_1 = \frac{1}{4}a, c_2 = \frac{7}{4}a, g(x) = x^2$
- ◆ $\frac{1}{1+x^2} = O(1), \quad \forall x \geq x_0 = 1$ **oraz np. dla $c \geq 1$;**
- ◆ $x! = O(x^x), \quad \forall x \geq x_0 = 1$ **oraz np. dla $c \geq 1$.**

SZACOWANIE ZŁOŻONOŚCI OBliczeniowej

Własności funkcji rzędów złożoności obliczeniowej – potencjalne problemy

Celem wprowadzonych wcześniej sposobów zapisu (notacji) jest porównanie efektywności rozmaitych algorytmów zaprojektowanych do rozwiązania tego samego problemu.

Jeżeli będziemy stosować tylko notacje „wielkie O” do reprezentowania złożoności algorytmów, to niektóre z nich możemy zdyskwalifikować zbyt pochopnie.

Przykład 3:

Załóżmy, że mamy dwa algorytmy rozwiązujące pewien problem, wykonywana przez nie liczba operacji to odpowiednio $10^8 n$ i $10n^2$. Pierwsza funkcja jest $O(n)$, druga $O(n^2)$. Opierając się na informacji dostarczonej przez notacje „wielkie O” odrzucilibyśmy drugi algorytm ponieważ funkcja kosztu rośnie zbyt szybko. To prawda ale dopiero dla odpowiednio dużych n , ponieważ dla $n < 10^7$ drugi algorytm wykonuje mniej operacji niż pierwszy.

Istotna jest więc też stała (10^8), która w tym przypadku jest zbyt duża, aby notacja była znacząca.

SZACOWANIE ZŁOŻONOŚCI OBliczeniowej

Jak mierzyć czas działania algorytmu?

Sposób pomiaru ilości pracy wykonanej przez algorytm powinien zapewniać:

- ◆ porównanie efektywności dwóch różnych algorytmów rozwiązujących ten sam problem;
- ◆ oszacowanie faktycznej wydajności metody, abstrahując od komputera, języka programowania, umiejętności programisty i szczegółów technicznych implementacji (sposobu inkrementacji zmiennych sterujących pętli, sposobu obliczania indeksów zmiennych ze wskaźnikami itp.);

Wniosek:

Dobrym przybliżeniem czasochłonności algorytmu jest obliczenie liczby przejść przez wszystkie pętle w algorytmie.

SZACOWANIE ZŁOŻONOŚCI OBliczeniowej

Jak mierzyć czas działania algorytmu?, c.d.

W praktyce zlicza się tzw. operacje podstawowe dla badanego problemu lub klasy rozważanych algorytmów, tj. takie, które są najczęściej wykonywane (ignorując pozostałe operacje pomocnicze, takie jak instrukcje inicjalizacji, instrukcje organizacji pętli itp.).

Ponieważ większość programów to programy złożone z wielu modułów lub podprogramów, a w każdym takim podprogramie inną instrukcję może grać rolę operacji podstawowej, więc fragmenty większej całości analizuje się zwykle oddzielnie i na podstawie skończonej liczby takich modułów szacuje się czasochłonność algorytmu jako całości.

SZACOWANIE ZŁOŻONOŚCI OBliczeniowej

Jak mierzyć czas działania algorytmu?, c.d.

Tabela 1 Przykłady operacji podstawowych dla typowych problemów obliczeniowych

Lp.	Problem	Operacja
1.	Znalezienie x na liście nazwisk.	Porównanie x z pozycją na liście.
2.	Mnożenie dwóch macierzy liczb rzeczywistych.	Mnożenie dwóch macierzy liczb typu real (lub mnożenie i dodawanie).
3.	Porządkowanie liczb.	Porównanie dwóch liczb (lub porównanie i zamiana).
4.	Wyznaczanie drogi najkrótszej w grafie zadanym w postaci listy sąsiadów.	Operacja na wskaźniku listy.

SZACOWANIE ZŁOŻONOŚCI OBliczeniowej

Jak mierzyć czas działania algorytmu?, c.d.

Zalety zliczania operacji podstawowych:

- ◆ możliwość przewidywania zachowania się algorytmu dla dużych rozmiarów danych (jeżeli łączna liczba operacji jest proporcjonalna do liczby operacji podstawowych);
- ◆ swoboda wyboru operacji podstawowej.

W skrajnym przypadku można wybrać rozkazy maszynowe konkretnego komputera. Z drugiej strony jedno przejście przez instrukcję pętli można również potraktować jako operację podstawową. W ten sposób możemy manipulować stopniem precyzji w zależności od potrzeb.

SZACOWANIE ZŁOŻONOŚCI OBLICZENIOWEJ

Przykłady praktyczne mierzenia złożoności obliczeniowej – Przykład 4

Przykład 4: Prosta pętla

```
for (i=sum=0; i<n; i++) sum+=a[i];
```

Powyższa pętla powtarza się n razy, podczas każdego jej przebiegu realizuje dwa przypisania: aktualizujące zmienną „sum” i zmianę wartości zmiennej „i”. Mamy zatem $2n$ przypisań podczas całego wykonania pętli; jej asymptotyczna złożoność wynosi $O(n)$.

SZACOWANIE ZŁOŻONOŚCI OBLICZENIOWEJ

Przykłady praktyczne mierzenia złożoności obliczeniowej – Przykład 5

Przykład 5: Pętla zagnieźdzona

```
for (i=0; i<n; i++) {  
    for (j=1, sum=a[0]; j<=i; j++)  
        sum+=a[j]; }
```

Na samym początku zmiennej „i” nadawana jest wartość początkowa. Pętla zewnętrzna powtarza się n razy, a w każdej jej iteracji wykonuje się wewnętrzna pętla oraz instrukcja przypisania wartości zmiennym „i”, „j”, „sum”. Pętla wewnętrzna wykonuje się „i” razy dla każdego $i \in \{1, \dots, n-1\}$, a na każdą iterację przypadają dwa przypisania: jedno dla „sum”, jedno dla „j”. Mamy zatem

$1 + 3n + 2(1+2+\dots+n-1) = 1 + 3n + n(n-1) = O(n) + O(n^2) = O(n^2)$
przypisań wykonywanych w całym programie. Jej asymptotyczna złożoność wynosi $O(n^2)$.

Pętle zagnieździone mają zwykle większą złożoność niż pojedyncze, jednak nie musi tak być zawsze.

SZACOWANIE ZŁOŻONOŚCI OBLICZENIOWEJ

Przykłady praktyczne mierzenia złożoności obliczeniowej – Przykład 6

Analiza tych dwóch przypadków była stosunkowo prosta ponieważ liczba iteracji nie zależała od wartości elementów tablicy.

Wyznaczenie złożoności asymptotycznej jest trudniejsze jeżeli liczba iteracji nie jest zawsze jednakowa.

Przykład 6: Znajdź najdłuższą podtablicę zawierającą liczby uporządkowane rosnąco.

```
for (i=0; len=1; i<n-1; i++) {  
    for (i1=i2=k=i; k<n-1 && a[k]<a[k+1]; k++,i2++);  
    if(len < i2-i1+1) len=i2-i1+1; }
```

=> Jeżeli liczby w tablicy są uporządkowane malejąco, to pętla zewnętrzna wykonuje się n-1 razy, a w każdym jej przebiegu pętla wewnętrzna wykona się tylko raz. Złożoność algorytmu jest więc $O(n)$.

SZACOWANIE ZŁOŻONOŚCI OBLICZENIOWEJ

Przykłady praktyczne mierzenia złożoności obliczeniowej – Przykład 6, c.d.

Analiza tych dwóch przypadków była stosunkowo prosta ponieważ liczba iteracji nie zależała od wartości elementów tablicy.

Wyznaczenie złożoności asymptotycznej jest trudniejsze jeżeli liczba iteracji nie jest zawsze jednakowa.

Przykład 6: Znajdź najdłuższą podtablicę zawierającą liczby uporządkowane rosnąco.

```
for (i=0; len=1; i<n-1; i++) {  
    for (i1=i2=k=i; k<n-1 && a[k]<a[k+1]; k++,i2++);  
    if(len < i2-i1+1) len=i2-i1+1; }
```

=> Jeżeli liczby w tablicy są uporządkowane rosnąco, to pętla zewnętrzna wykonuje się n-1 razy, a w każdym jej przebiegu pętla wewnętrzna wykona się i razy dla $i \in \{1, \dots, n-1\}$.
Złożoność algorytmu jest więc $O(n^2)$.

SZACOWANIE ZŁOŻONOŚCI OBliczeniowej

Przykłady praktyczne mierzenia złożoności obliczeniowej – Przykład 6, c.d.

Analiza tych dwóch przypadków była stosunkowo prosta ponieważ liczba iteracji nie zależała od wartości elementów tablicy.

Wyznaczenie złożoności asymptotycznej jest trudniejsze jeżeli liczba iteracji nie jest zawsze jednakowa.

Przykład 6: Znajdź najdłuższą podtablicę zawierającą liczby uporządkowane rosnąco.

```
for (i=0; len=1; i<n-1; i++) {  
    for (i1=i2=k=i; k<n-1 && a[k]<a[k+1]; k++,i2++);  
        if(len < i2-i1+1) len=i2-i1+1; }
```

=> Z reguły dane nie są uporządkowane i ocena złożoności algorytmu jest rzeczą niełatwą, ale bardzo istotną. Staramy się wyznaczyć złożoność w „przypadku optymistycznym”, „przypadku pesymistycznym” oraz „przypadku średnim”. Często posługujemy się przybliżeniami opartymi o wcześniej zdefiniowane notacje „duże O, Ω i Θ ”.



Dziękuję za uwagę

Algorytmy i Struktury Danych.

Treści programowe. Złożoność obliczeniowa algorytmu na przykładach.

dr hab. Bożena Woźna-Szcześniak
bwozna@gmail.com

Jan Długosz University, Poland

Wykład 1



Cel wykładów

- Zapoznanie z podstawowym zestawem algorytmów realizujących zadania typu wyszukiwanie, sortowanie.
- Zapoznanie z najczęściej wykorzystywanymi abstrakcyjnymi strukturami danych: stosami, kolejkami, listami, kolejkami priorytetowymi, grafami i drzewami.
- Zapoznanie z podstawowymi algorytmami grafowymi.
- Zapoznanie z podstawami analizy algorytmów – dobór właściwego algorytmu i struktury danych jest podstawą sukcesu przy rozwiązywaniu zadanego problemu.
- Zapoznanie z technikami projektowania algorytmów: dziel i rządź, programowanie dynamiczne, algorytmy zachłanne, rekurencja.

Treści programowe: Wykład

- Złożoność obliczeniowa algorytmu na przykładach.
- Podstawowe techniki projektowania algorytmów: rekurencja.
- Podstawowe techniki projektowania algorytmów: metoda dziel i zwyciężaj. Przykładowe algorytmy: znajdowanie największego i najmniejszego elementu zbioru, mnożenie liczb całkowitych, wyszukiwanie binarne, sortowanie szybkie, sortowanie przez scalanie.
- Pojęcie stabilności dla problemu sortowania. Algorytmy sortujące: sortowanie przez zliczanie i pozycyjne.
- Abstrakcyjne struktury danych i ich implementacje: listy dwu- i jednokierunkowe, cykliczne listy dwu- i jednokierunkowe, stos i kolejka, odwrotna notacja polska.
- Abstrakcyjne struktury danych i ich implementacje: drzewa poszukiwań binarnych.

Treści programowe: Wykład - cd.

- Abstrakcyjne struktury danych i ich implementacje: kopce binarne i kolejki priorytetowe, sortowanie przez kopcowanie.
- Grafy i metody ich reprezentacji. Algorytm DFS i BFS – przykładowa implementacja.
- Algorytmy grafowe: wyszukiwanie spójnych składowych grafu, sortowanie topologiczne, cykle Eulera, problemy ścieżkowe (algorytm Bellmana-Forda, algorytm Dijkstry, Floyd-Warshall).
- Algorytmy zachłanne: minimalne drzewo rozpinające – ciągły problem plecakowy, algorytm Kruskala, algorytm Prima; Kodowanie Huffmana, pokrycie zbioru.
- Programowanie dynamiczne: dyskretny problem plecakowy, mnożenie łańcucha macierzy, najdłuższy podciąg rosnący, odległość edycyjna.
- Wyszukiwanie wzorca w tekstach: prefikso-sufiksy, algorytm Knutha-Morisa-Pratta, algorytm Boyera-Moora.

Treści programowe: Laboratorium

- Implementacja algorytmów oraz problemów algorytmicznych prezentowanych na wykładzie w wybranym języku programowania.
- Analiza złożoności wykonywanych implementacji.

- Cormen T.H., Leiserson Ch.E., Rivest R.L. Wprowadzenie do algorytmów. WNT, Warszawa, 1997 i późniejsze.
- Dasgupta S., Papadimitriou Ch., Vazirani U. Algorytmy. Seria Fundamenty Informatyki, Wydawnictwo Naukowe PWN, Warszawa 2010.
- Banachowski L., Diks K., Rytter W. Algorytmy i struktury danych. WNT, Warszawa, 1996.

Na całym cyklu wykładów z Algorytmów i Struktur Danych wykorzystano materiały dostępne z następujących witryn:

- <http://wazniak.mimuw.edu.pl/>
- <http://ocw.mit.edu/courses/>

Dlaczego warto poznać algorytmy i ich złożoność obliczeniową ?

- Algorytmy pozwalają na pisanie **dobrych programów**, tzn. programów, które działają poprawnie i wydajne, są funkcjonalne, zużywają tylko konieczną liczbę zasobów, itp.

Dlaczego warto poznać algorytmy i ich złożoność obliczeniową ?

- Algorytmy pozwalają na pisanie **dobrych programów**, tzn. programów, które działają poprawnie i wydajne, są funkcjonalne, zużywają tylko konieczną liczbę zasobów, itp.
- Algorytmy pozwalają w sposób formalny mówić o zachowaniu programów implementowanych w różnych językach programowania.

Dlaczego warto poznać algorytmy i ich złożoność obliczeniową ?

- Algorytmy pozwalają na pisanie **dobrych programów**, tzn. programów, które działają poprawnie i wydajne, są funkcjonalne, zużywają tylko konieczną liczbę zasobów, itp.
- Algorytmy pozwalają w sposób formalny mówić o zachowaniu programów implementowanych w różnych językach programowania.
- Najważniejszymi aspektami algorytmu są jego **poprawność** i **złożoność obliczeniowa**.

Dlaczego warto poznać algorytmy i ich złożoność obliczeniową ?

- Algorytmy pozwalają na pisanie **dobrych programów**, tzn. programów, które działają poprawnie i wydajne, są funkcjonalne, zużywają tylko konieczną liczbę zasobów, itp.
- Algorytmy pozwalają w sposób formalny mówić o zachowaniu programów implementowanych w różnych językach programowania.
- Najważniejszymi aspektami algorytmu są jego **poprawność** i **złożoność obliczeniowa**.
- Złożoność obliczeniowa algorytmu wyznacza granice pomiędzy tym co jest możliwe do wykonania/zimplementowania/zastosowania w rzeczywistości, a co nie.

Dlaczego warto poznać algorytmy i ich złożoność obliczeniową ?

- Algorytmy pozwalają na pisanie **dobrych programów**, tzn. programów, które działają poprawnie i wydajne, są funkcjonalne, zużywają tylko konieczną liczbę zasobów, itp.
- Algorytmy pozwalają w sposób formalny mówić o zachowaniu programów implementowanych w różnych językach programowania.
- Najważniejszymi aspektami algorytmu są jego **poprawność** i **złożoność obliczeniowa**.
- Złożoność obliczeniowa algorytmu wyznacza granice pomiędzy tym co jest możliwe do wykonania/zimplementowania/zastosowania w rzeczywistości, a co nie.
- **Złożoność obliczeniowa** algorytmu to **koszt** jego realizacji.

Złożoność algorytmu

Definicja

Złożoność algorytmu to ilość zasobów komputera niezbędnych do jego wykonania. W zależności od rozważanego zasobu złożoność dzielimy na **złożoność czasową** oraz **złożoność pamięciową**.

Złożoność algorytmu

Definicja

Złożoność algorytmu to ilość zasobów komputera niezbędnych do jego wykonania. W zależności od rozważanego zasobu złożoność dzielimy na **złożoność czasową** oraz **złożoność pamięciową**.

- **Złożoność czasowa** to zależność pomiędzy rozmiarem danych wejściowych a liczbą operacji elementarnych (operacji dominujących) wykonywanych w trakcie przebiegu algorytmu.

Operacjami elementarnymi mogą być na przykład: *podstawienie*, *porównanie* lub *prosta operacja arytmetyczna*. Dzięki rozważaniu operacji dominujących analiza złożoności jest zależna jedynie od algorytmu, a nie od jego implementacji i sprzętu.

Złożoność algorytmu

- **Złożoność pamięciowa** to zależność pomiędzy rozmiarem danych wejściowych a ilością wykorzystanej pamięci. Jako tę ilość najczęściej przyjmuje się.used memory of the machine (for example, the number of memory cells of the RAM) in function of the size of the input.

Złożoność algorytmu

- **Złożoność pamięciowa** to zależność pomiędzy rozmiarem danych wejściowych a ilością wykorzystanej pamięci. Jako tę ilość najczęściej przyjmuje się.used pamięć maszyny abstrakcyjnej (na przykład liczbę komórek pamięci maszyny RAM) w funkcji rozmiaru wejścia.
- Im większe rozmiary danych wejściowych tym więcej zasobów (czasu, pamięci) jest koniecznych do wykonania danego algorytmu. Złożoność algorytmu jest zatem funkcją rozmiaru danych wejściowych.

Złożoność algorytmu

- **Złożoność pamięciowa** to zależność pomiędzy rozmiarem danych wejściowych a ilością wykorzystanej pamięci. Jako tę ilość najczęściej przyjmuje się.used memory of a machine (for example, the number of RAM memory cells) in function of the size of the input data.
- Im większe rozmiary danych wejściowych tym więcej zasobów (czasu, pamięci) jest koniecznych do wykonania danego algorytmu. Złożoność algorytmu jest zatem funkcją rozmiaru danych wejściowych.
- Złożoność algorytmu zależy nie tylko od rozmiaru ciągu wejściowego (krótkie ciągi są znacznie łatwiej posortować niż długie) ale również od rodzaju wejścia (prawie uporządkowany ciąg jest znacznie łatwiej posortować).

Rodzaje złożoności

W praktyce rozważa się dwa podejścia: rozpatrywanie przypadków najgorszych (**złożoność pesymistyczna**) oraz zastosowanie określonego sposobu uśrednienia wszystkich możliwych przypadków (**złożoność oczekiwana**).

- Złożoność pesymistyczna (ang. worst-case):
 - $T(n)$ = maksymalna ilość zasobu (pamięć, czas) potrzebna do wykonania algorytmu dla dowolnego wejścia o rozmiarze n .
- Złożoność oczekiwana (ang. average-case):
 - $T(n)$ = oczekiwana ilość zasobu (pamięć, czas) potrzebna do wykonania algorytmu dla dowolnego wejścia o rozmiarze n .
 - Złożoność oczekiwana zależy istotnie od założenia o rozważanej przestrzeni probabilistycznej danych wejściowych.
 - Przestrzeń probabilistyczna danych wejściowych może być bardzo skomplikowana, co powoduje, że wyznaczenie złożoności oczekiwanej wymaga bardzo trudnych analiz matematycznych.

Notacja O wielkie

Niech $f(n)$ oraz $g(n)$ będą funkcjami ze bioru liczb naturalnych w zbiór liczb rzeczywistych (tj. $f : \mathbb{N} \rightarrow \mathbb{R}$, $g : \mathbb{N} \rightarrow \mathbb{R}$), które są czasami przebiegu dwóch algorytmów działających na danych wejściowych o rozmiarze n (lub które reprezentują ilość pamięci wykorzystanej przez dwa alorytmy działające na danych wejściowych o rozmiarze n).

Powiemy, że f **rośnie nie szybciej niż** g i oznaczamy:

$$f(n) = O(g(n))$$

jeżeli istnieją stałe $c > 0$ oraz $n_0 \in \mathbb{N}$ takie, że $f(n) \leq c \cdot g(n)$ dla wszystkich $n \geq n_0$

Notacja O wielkie

Powiemy, że f jest niemniejsza niż g i oznaczamy $f(n) = \Omega(g(n))$, jeżeli istnieją stałe $c > 0$ oraz $n_0 \in \mathbb{N}$ takie, że $f(n) \geq c \cdot g(n)$ dla wszystkich $n \geq n_0$.

Notacja O wielkie

Powiemy, że f jest niemniejsza niż g i oznaczamy $f(n) = \Omega(g(n))$, jeżeli istnieją stałe $c > 0$ oraz $n_0 \in \mathbb{N}$ takie, że $f(n) \geq c \cdot g(n)$ dla wszystkich $n \geq n_0$.

Uwaga!

$f(n) = \Omega(g(n))$ oznacza, że $g(n) = O(f(n))$

Notacja O wielkie

Powiemy, że f jest niemniejsza niż g i oznaczamy $f(n) = \Omega(g(n))$, jeżeli istnieją stałe $c > 0$ oraz $n_0 \in \mathbb{N}$ takie, że $f(n) \geq c \cdot g(n)$ dla wszystkich $n \geq n_0$.

Uwaga!

$f(n) = \Omega(g(n))$ oznacza, że $g(n) = O(f(n))$

Powiemy, że f jest podobna do g i oznaczamy $f(n) = \Theta(g(n))$, jeżeli istnieją stałe $c_0, c_1 > 0$ i $n_0 \in \mathbb{N}$, takie, że $c_0 \cdot g(n) \leq f(n) \leq c_1 \cdot g(n)$ dla wszystkich $n \geq n_0$

Notacja O wielkie

Powiemy, że f jest niemniejsza niż g i oznaczamy $f(n) = \Omega(g(n))$, jeżeli istnieją stałe $c > 0$ oraz $n_0 \in \mathbb{N}$ takie, że $f(n) \geq c \cdot g(n)$ dla wszystkich $n \geq n_0$.

Uwaga!

$f(n) = \Omega(g(n))$ oznacza, że $g(n) = O(f(n))$

Powiemy, że f jest podobna do g i oznaczamy $f(n) = \Theta(g(n))$, jeżeli istnieją stałe $c_0, c_1 > 0$ i $n_0 \in \mathbb{N}$, takie, że $c_0 \cdot g(n) \leq f(n) \leq c_1 \cdot g(n)$ dla wszystkich $n \geq n_0$

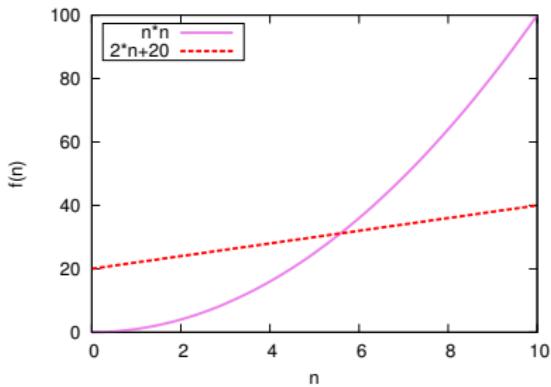
Uwaga!

$f(n) = \Theta(g(n))$ oznacza, że $f(n) = O(g(n))$ oraz $f(n) = \Omega(g(n))$

Notacja O wielkie - Przykład 1

Założymy, że mamy wybrać jeden z dwóch algorytmów rozwiązujących pewien zadany problem obliczeniowy.

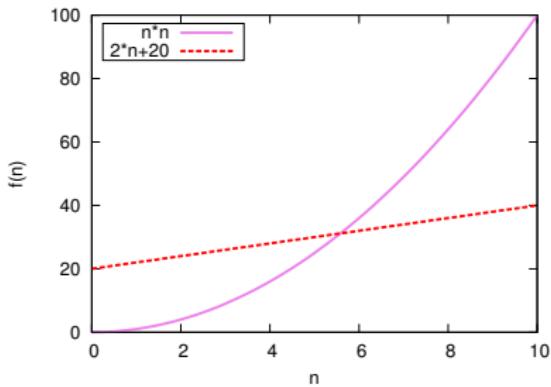
- Pierwszy algorytm wykonuje $f_1(n) = n^2$ kroków.
- Drugi algorytm wykonuje $f_2(n) = 2n + 20$ kroków.



Notacja O wielkie - Przykład 1

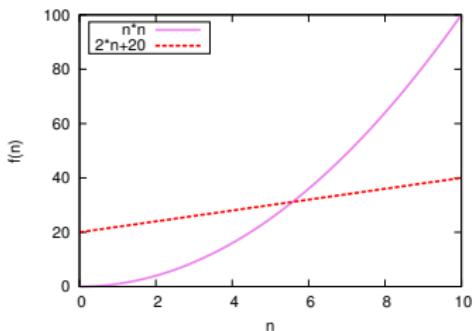
Założymy, że mamy wybrać jeden z dwóch algorytmów rozwiązujących pewien zadany problem obliczeniowy.

- Pierwszy algorytm wykonuje $f_1(n) = n^2$ kroków.
- Drugi algorytm wykonuje $f_2(n) = 2n + 20$ kroków.



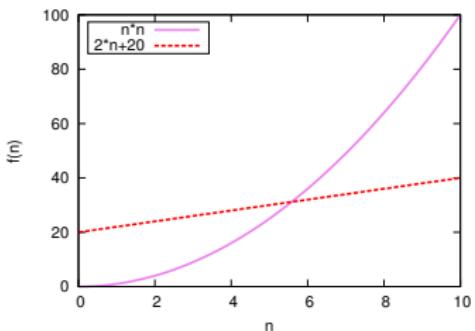
Który z nich jest lepszy ?

Notacja O wielkie - Przykład 1, cd.



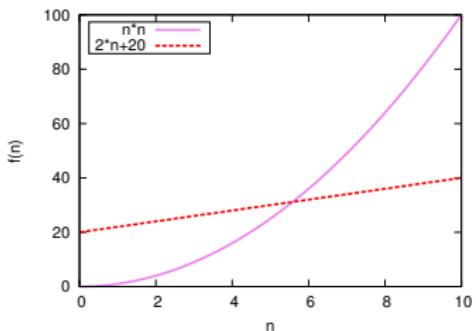
- Odpowiedź zależy od wartości n !

Notacja O wielkie - Przykład 1, cd.



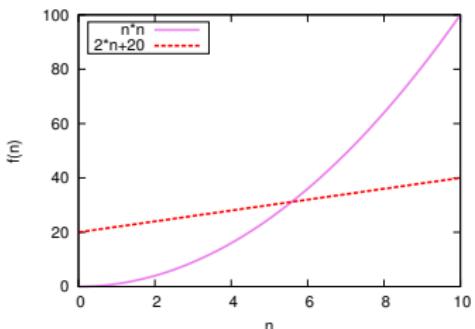
- Odpowiedź zależy od wartości n !
- Dla $n \leq 5$ wartości n^2 są mniejsze od $2n + 20$.
- Dla $n > 5$ wartości n^2 są większe od $2n + 20$.

Notacja O wielkie - Przykład 1, cd.



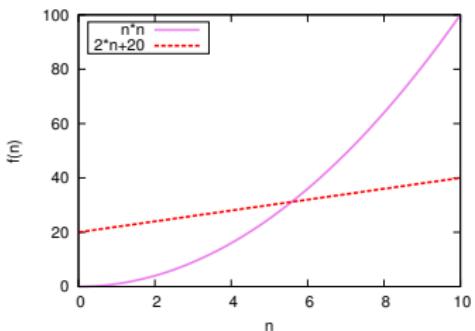
- Odpowiedź zależy od wartości n !
- Dla $n \leq 5$ wartości n^2 są mniejsze od $2n + 20$.
- Dla $n > 5$ wartości n^2 są większe od $2n + 20$.
- **Wniosek !**

Notacja O wielkie - Przykład 1, cd.



- Odpowiedź zależy od wartości n !
- Dla $n \leq 5$ wartości n^2 są mniejsze od $2n + 20$.
- Dla $n > 5$ wartości n^2 są większe od $2n + 20$.
- **Wniosek !** Funkcja $f_2(n) = 2n + 20$ zachowuje się znacznie lepiej, gdy n rośnie. Dlatego też algorytm drugi jest **zwycięzcą**.

Notacja O wielkie - Przykład 1, cd.



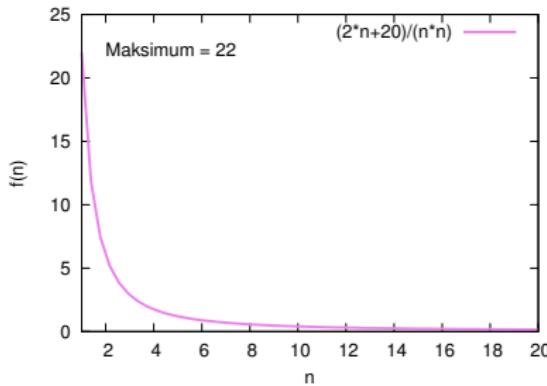
- Odpowiedź zależy od wartości n !
- Dla $n \leq 5$ wartości n^2 są mniejsze od $2n + 20$.
- Dla $n > 5$ wartości n^2 są większe od $2n + 20$.
- **Wniosek !** Funkcja $f_2(n) = 2n + 20$ zachowuje się znacznie lepiej, gdy n rośnie. Dlatego też algorytm drugi jest zwycięzcą.
- Powyższą własność można opisać przez notację **O wielkie**:
 $f_2(n) = O(f_1(n))$.

Notacja O wielkie - Przykład 1, cd.

Dlaczego $f_2(n) = O(f_1(n))$?

Ponieważ dla wszystkich $n > 0$ zachodzi:

$$\frac{f_2(n)}{f_1(n)} = \frac{2n + 20}{n^2} \leq 22$$



tzn.

$$f_2(n) \leq 22 \cdot f_1(n)$$

Notacja O wielkie - Przykład 1, cd.

Dlaczego $f_1(n) \neq O(f_2(n))$?

Ponieważ stosunek

$$\frac{f_1(n)}{f_2(n)} = \frac{n^2}{2n + 20}$$

Notacja O wielkie - Przykład 1, cd.

Dlaczego $f_1(n) \neq O(f_2(n))$?

Ponieważ stosunek

$$\frac{f_1(n)}{f_2(n)} = \frac{n^2}{2n + 20}$$

może być dowolnie duży !

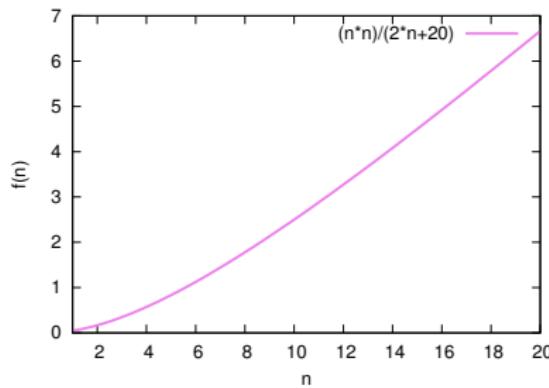
Notacja O wielkie - Przykład 1, cd.

Dlaczego $f_1(n) \neq O(f_2(n))$?

Ponieważ stosunek

$$\frac{f_1(n)}{f_2(n)} = \frac{n^2}{2n + 20}$$

może być dowolnie duży !



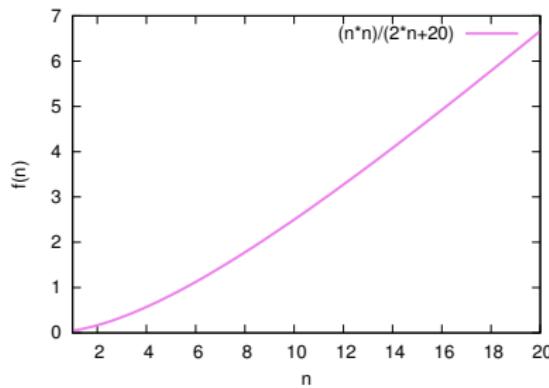
Notacja O wielkie - Przykład 1, cd.

Dlaczego $f_1(n) \neq O(f_2(n))$?

Ponieważ stosunek

$$\frac{f_1(n)}{f_2(n)} = \frac{n^2}{2n + 20}$$

może być dowolnie duży !



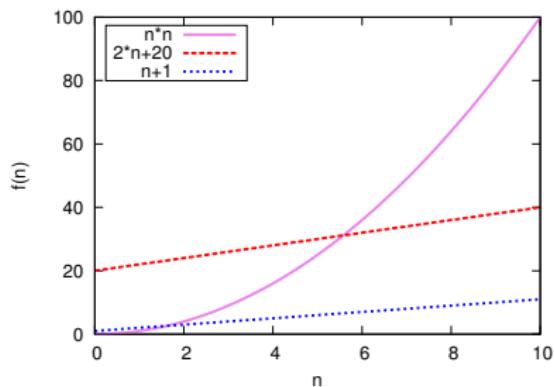
Nie istnieje zatem żadna stała c spełniająca warunki definicji.

Notacja O wielkie - Przykład 1, cd.

Przyjmijmy teraz, że mamy także trzeci algorytm rozwiązujejący zadany problem, przykładowo, w $f_3(n) = n + 1$ krokach.

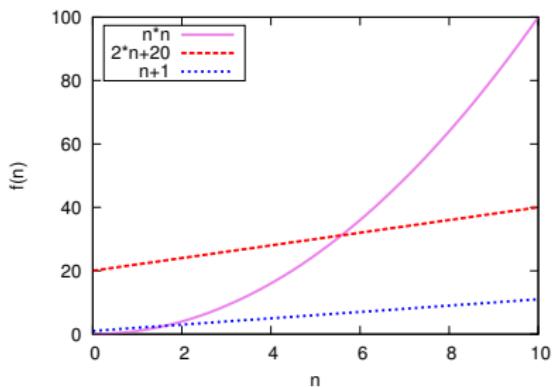
Notacja O wielkie - Przykład 1, cd.

Przyjmijmy teraz, że mamy także trzeci algorytm rozwiązujący zadany problem, przykładowo, w $f_3(n) = n + 1$ krokach.



Notacja O wielkie - Przykład 1, cd.

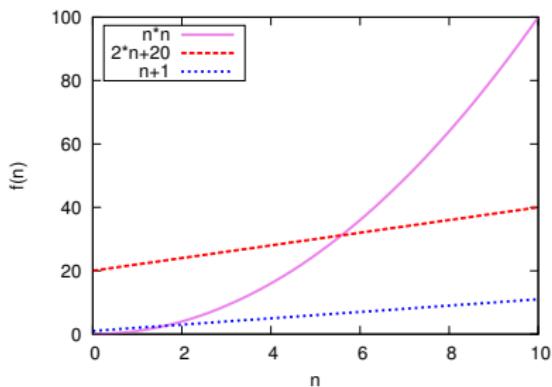
Przyjmijmy teraz, że mamy także trzeci algorytm rozwiązujący zadany problem, przykładowo, w $f_3(n) = n + 1$ krokach.



Czy algorytm trzeci jest lepszy od algorytmu drugiego ?

Notacja O wielkie - Przykład 1, cd.

Przyjmijmy teraz, że mamy także trzeci algorytm rozwiązujejący zadany problem, przykładowo, w $f_3(n) = n + 1$ krokach.



Czy algorytm trzeci jest lepszy od algorytmu drugiego ? **TAK**, ale tylko z dokładnością do stałego współczynnika.

Notacja O wielkie - Przykład 1, cd.

Zachodzi: $f_2(n) = O(f_3(n))$.

Dlaczego:

Notacja O wielkie - Przykład 1, cd.

Zachodzi: $f_2(n) = O(f_3(n))$.

Dlaczego: ponieważ dla wszystkich $n \geq 0$ zachodzi:

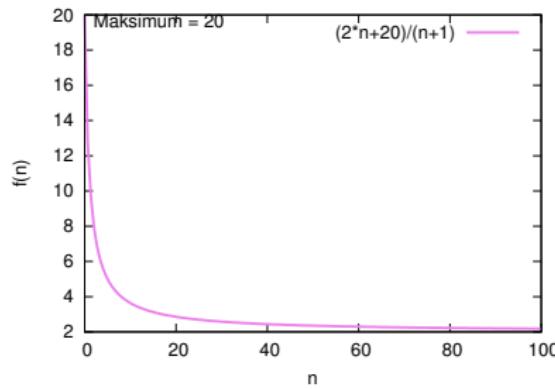
$$\frac{f_2(n)}{f_3(n)} = \frac{2n + 20}{n + 1} \leq 20$$

Notacja O wielkie - Przykład 1, cd.

Zachodzi: $f_2(n) = O(f_3(n))$.

Dlaczego: ponieważ dla wszystkich $n \geq 0$ zachodzi:

$$\frac{f_2(n)}{f_3(n)} = \frac{2n + 20}{n + 1} \leq 20$$

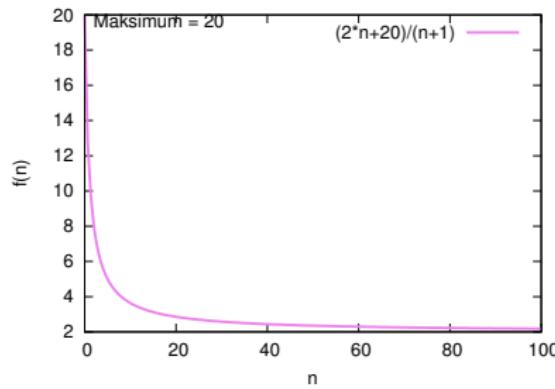


Notacja O wielkie - Przykład 1, cd.

Zachodzi: $f_2(n) = O(f_3(n))$.

Dlaczego: ponieważ dla wszystkich $n \geq 0$ zachodzi:

$$\frac{f_2(n)}{f_3(n)} = \frac{2n + 20}{n + 1} \leq 20$$



tzn.

$$f_2(n) \leq 20 \cdot f_3(n)$$

Notacja O wielkie - Przykład 1, cd.

Zachodzi: $f_3(n) = O(f_2(n))$.

Dlaczego:

Notacja O wielkie - Przykład 1, cd.

Zachodzi: $f_3(n) = O(f_2(n))$.

Dlaczego: ponieważ dla wszystkich $n \geq 0$ zachodzi:

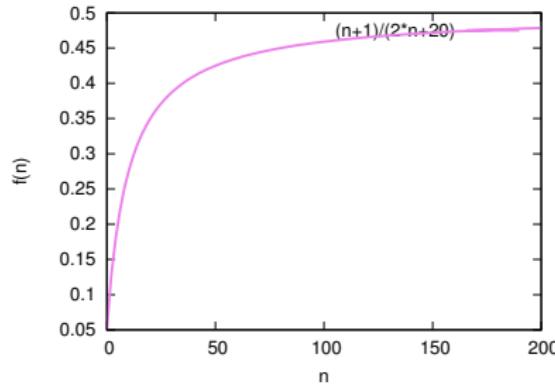
$$\frac{f_3(n)}{f_2(n)} = \frac{n+1}{2n+20} \leq 1$$

Notacja O wielkie - Przykład 1, cd.

Zachodzi: $f_3(n) = O(f_2(n))$.

Dlaczego: ponieważ dla wszystkich $n \geq 0$ zachodzi:

$$\frac{f_3(n)}{f_2(n)} = \frac{n+1}{2n+20} \leq 1$$

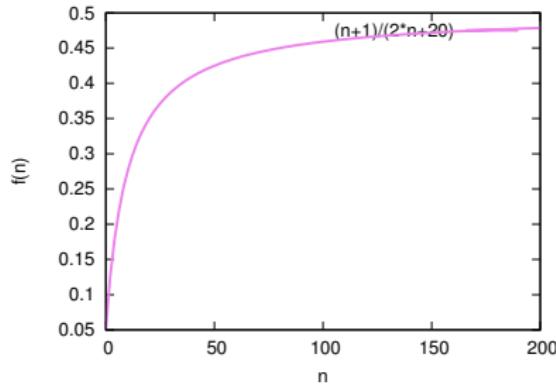


Notacja O wielkie - Przykład 1, cd.

Zachodzi: $f_3(n) = O(f_2(n))$.

Dlaczego: ponieważ dla wszystkich $n \geq 0$ zachodzi:

$$\frac{f_3(n)}{f_2(n)} = \frac{n+1}{2n+20} \leq 1$$



tzn.

$$f_3(n) \leq 1 \cdot f_2(n)$$

Wnioski z przykładu !

- A. $f_2(n) = O(f_1(n))$, ale $f_1(n) \neq O(f_2(n))$
- B. $f_3(n) = O(f_1(n))$, ale $f_1(n) \neq O(f_3(n))$
- C. $f_2(n) = O(f_3(n))$ oraz $f_3(n) = O(f_2(n))$

Wnioski z przykładu !

- A. $f_2(n) = O(f_1(n))$, ale $f_1(n) \neq O(f_2(n))$
- B. $f_3(n) = O(f_1(n))$, ale $f_1(n) \neq O(f_3(n))$
- C. $f_2(n) = O(f_3(n))$ oraz $f_3(n) = O(f_2(n))$

Z definicji Ω mamy:

- A. $f_1(n) = \Omega(f_2(n))$
- B. $f_1(n) = \Omega(f_3(n))$
- C. $f_3(n) = \Omega(f_2(n))$ oraz $f_2(n) = \Omega(f_3(n))$

Notacja O wielkie - Przykład 1, cd.

Wnioski z przykładu !

- A. $f_2(n) = O(f_1(n))$, ale $f_1(n) \neq O(f_2(n))$
- B. $f_3(n) = O(f_1(n))$, ale $f_1(n) \neq O(f_3(n))$
- C. $f_2(n) = O(f_3(n))$ oraz $f_3(n) = O(f_2(n))$

Z definicji Ω mamy:

- A. $f_1(n) = \Omega(f_2(n))$
- B. $f_1(n) = \Omega(f_3(n))$
- C. $f_3(n) = \Omega(f_2(n))$ oraz $f_2(n) = \Omega(f_3(n))$

Z definicji Θ mamy:

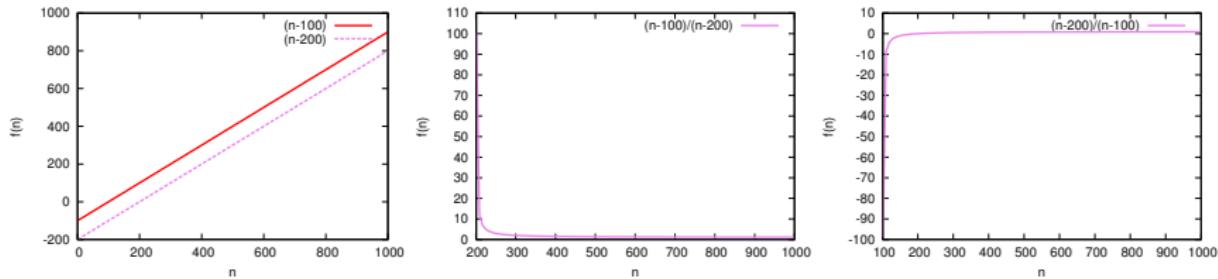
- A. $f_2(n) = \Theta(f_3(n))$ oraz $f_3(n) = \Theta(f_2(n))$

Notacja O wielkie - Zadania

Dane są następujące funkcje $f(n)$ oraz $g(n)$. Zadecyduj, które z sytuacji zachodzi: $f = O(g)$, $f = \Omega(g)$, czy też $f = \Theta(g)$

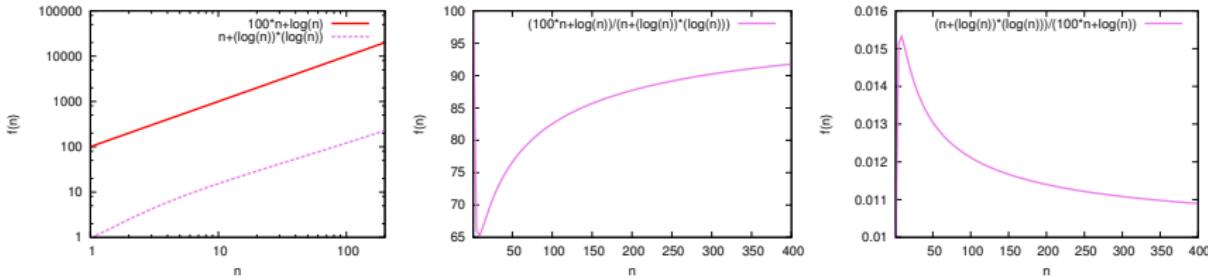
- (a) $f(n) = n - 100$, $g(n) = n - 200$
- (b) $f(n) = 100n + \log(n)$, $g(n) = n + (\log(n))^2$
- (c) $f(n) = n * 2^n$, $g(n) = 3^n$

Notacja O wielkie - Odpowiedzi



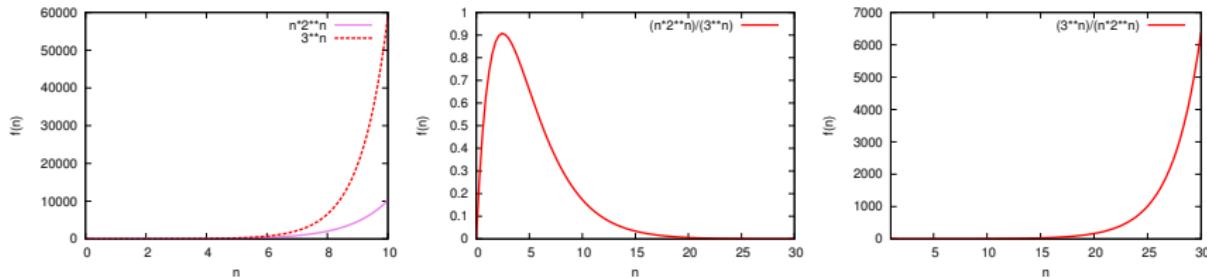
- (a) Ponieważ $\frac{f(n)}{g(n)} = \frac{n-100}{n-200} \leq 101$ dla wszystkich $n \geq 201$, to $f(n) = O(g(n))$.
Ponieważ stosunek $\frac{g(n)}{f(n)} = \frac{n-200}{n-100} \leq 1$ dla wszystkich $n \geq 101$, to $g(n) = O(f(n))$.
Z definicji Ω mamy, że $f(n) = \Omega(g(n))$ oraz $g(n) = \Omega(f(n))$.
Z definicji Θ mamy, że $f(n) = \Theta(g(n))$ oraz $g(n) = \Theta(f(n))$

Notacja O wielkie - Odpowiedzi



(b) Ponieważ stosunek $\frac{f(n)}{g(n)} = \frac{100n + \log(n)}{n + (\log(n))^2}$ może być dowolnie duży, to $f(n) \neq O(g(n))$. Ponieważ stosunek $\frac{g(n)}{f(n)} = \frac{n + (\log(n))^2}{100n + \log(n)} \leq 1$ dla wszystkich $n > 0$, to $g(n) = O(f(n))$. Z definicji Ω mamy, że $f(n) = \Omega(g(n))$.

Notacja O wielkie - Odpowiedzi



- (c) Ponieważ stosunek $\frac{f(n)}{g(n)} = \frac{n \cdot 2^n}{3^n} \leq 1$ dla wszystkich $n > 0$, to $f(n) = O(g(n))$.
Ponieważ stosunek $\frac{g(n)}{f(n)} = \frac{3^n}{n \cdot 2^n}$ może być dowolnie duży, to $g(n) \neq O(f(n))$.
Z definicji Ω mamy, że $g(n) = \Omega(f(n))$.

Notacja O wielkie

- Gdy $g(n) = \log(n)$, to mówimy, że $f(n)$ jest **logarytmiczna**.
- Gdy $g(n) = n$, to mówimy, że $f(n)$ jest **liniowa**.
- Gdy $g(n) = n * \log(n)$, to mówimy, że $f(n)$ jest **liniowo-logarytmiczna**.
- Gdy $g(n) = n^2$, to mówimy, że złożoność $f(n)$ jest **kwadratowa**.
- Jeśli $g(n)$ jest wielomianem, to mówimy o złożoności **wielomianowej** dla $f(n)$.
- Jeśli $g(n)$ jest wykładnicza, to mówimy o złożoności **wykładniczej** dla $f(n)$.
- Jeśli $g(n)$ jest silnie wykładnicza, to mówimy o złożoności **silnie wykładniczej** dla $f(n)$.

Algorytm bisekcji (A,n,x):

```
1: i := 0; j := n – 1;  
2: while (j – i > 1) do  
3:   m := (i + j)div2;  
4:   if A[m] ≤ x then  
5:     i := m;  
6:   else  
7:     j := m;  
8:   end if  
9: end while  
10: if A[i] = x then  
11:   return true;  
12: else  
13:   return false;  
14: end if
```

- Wyszukiwanie binarne/algorytm bisekcji.

Algorytm bisekcji (A, n, x):

```
1: i := 0; j := n – 1;  
2: while (j – i > 1) do  
3:   m := (i + j)div2;  
4:   if A[m] ≤ x then  
5:     i := m;  
6:   else  
7:     j := m;  
8:   end if  
9: end while  
10: if A[i] = x then  
11:   return true;  
12: else  
13:   return false;  
14: end if
```

- Wyszukiwanie binarne/algorytm bisekcji.
- **A** oznacza uporządkowany ciąg elementów, **n** oznacza ilość elementów w ciągu, **x** oznacza poszukiwany element.

Algorytm bisekcji (A, n, x):

```
1:  $i := 0; j := n - 1;$ 
2: while ( $j - i > 1$ ) do
3:    $m := (i + j) \text{div} 2;$ 
4:   if  $A[m] \leq x$  then
5:      $i := m;$ 
6:   else
7:      $j := m;$ 
8:   end if
9: end while
10: if  $A[i] = x$  then
11:   return true;
12: else
13:   return false;
14: end if
```

- Wyszukiwanie binarne/algorytm bisekcji.
- **A** oznacza uporządkowany ciąg elementów, **n** oznacza ilość elementów w ciągu, **x** oznacza poszukiwany element.
- Złożoność czasowa: $O(\log(n))$.

Algorytm bisekcji (**A**,**n**,**x**):

```
1: i := 0; j := n – 1;  
2: while (j – i > 1) do  
3:   m := (i + j)div2;  
4:   if A[m] ≤ x then  
5:     i := m;  
6:   else  
7:     j := m;  
8:   end if  
9: end while  
10: if A[i] = x then  
11:   return true;  
12: else  
13:   return false;  
14: end if
```

- Wyszukiwanie binarne/algorytm bisekcji.
- **A** oznacza uporządkowany ciąg elementów, **n** oznacza ilość elementów w ciągu, **x** oznacza poszukiwany element.
- Złożoność czasowa: $O(\log(n))$.
- Czas działania **logarytmiczny** występuje dla algorytmów typu: zadanie rozmiaru n zostaje sprowadzone do zadania rozmiaru $n/2$ plus pewna stała liczba działań.

Algorytm zwiększania liczby o jeden w systemie binarnym

Aby zwiększyć o jeden liczbę binarną należy:

- 1 Wskazać ostatni bit.
- 2 Jeśli wskazany bit jest zerem, to ustawiamy go na jeden i kończymy algorytm.
- 3 Jeśli wskazany bit jest jedynką, to zmieniamy go na zero i przesuwamy się o jeden bit w lewo.
- 4 Jeżeli nie jest to pierwszy bit liczby, to przechodzimy z powrotem do kroku 2,
- 5 Jeżeli jest to pierwszy bit liczby to stawiamy jedynkę na początku i kończymy algorytm.

Algorytm zwiększania liczby o jeden w systemie binarnym

Aby zwiększyć o jeden liczbę binarną należy:

- 1 Wskazać ostatni bit.
- 2 Jeśli wskazany bit jest zerem, to ustawiamy go na jeden i kończymy algorytm.
- 3 Jeśli wskazany bit jest jedynką, to zmieniamy go na zero i przesuwamy się o jeden bit w lewo.
- 4 Jeżeli nie jest to pierwszy bit liczby, to przechodzimy z powrotem do kroku 2,
- 5 Jeżeli jest to pierwszy bit liczby to stawiamy jedynkę na początku i kończymy algorytm.

Przykład:

- Zwiększamy liczbę $100101_{(2)}$ o 1: $100110_{(2)}$.

Przykładowe algorytmy i ich złożoności

Algorytm zwiększania liczby o jeden w systemie binarnym

Aby zwiększyć o jeden liczbę binarną należy:

- 1 Wskazać ostatni bit.
- 2 Jeśli wskazany bit jest zerem, to ustawiamy go na jeden i kończymy algorytm.
- 3 Jeśli wskazany bit jest jedynką, to zmieniamy go na zero i przesuwamy się o jeden bit w lewo.
- 4 Jeżeli nie jest to pierwszy bit liczby, to przechodzimy z powrotem do kroku 2,
- 5 Jeżeli jest to pierwszy bit liczby to stawiamy jedynkę na początku i kończymy algorytm.

Przykład:

- Zwiększamy liczbę $100101_{(2)}$ o 1: $100110_{(2)}$.
- Zwiększamy liczbę $111_{(2)}$ o 1: $1000_{(2)}$.

Złożoność czasowa

$O(n)$, gdzie n oznacza liczbę bitów.

Uwaga!

- Czas działania **liniowy** występuje dla algorytmów, w których jest wykonywana pewna stała liczba działań dla każdego z n elementów danych wejściowych.
- Czas działania **liniowo-logarytmiczny** występuje dla algorytmów typu: zadanie rozmiaru n zostaje sprowadzone do dwóch podzadań rozmiaru $n/2$ plus pewna liczba działań, liniowa względem rozmiaru n , potrzebnych najpierw do wykonania rozbicia, a następnie scalenia rozwiązań rozmiaru $n/2$ w rozwiązanie rozmiaru n .

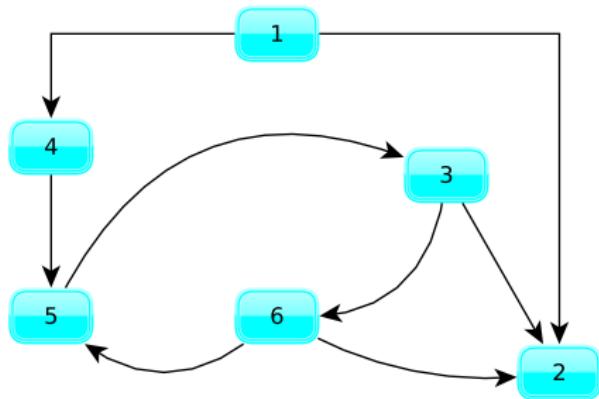
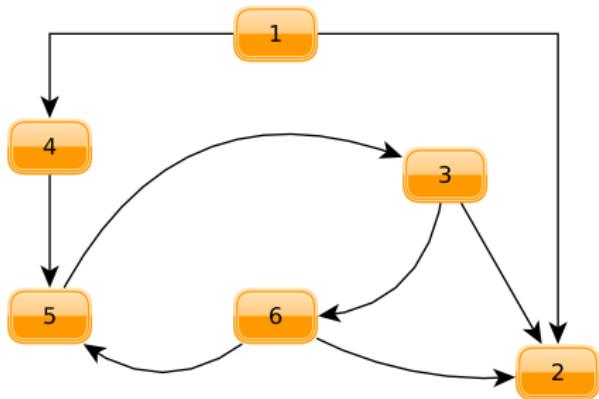
Uwaga!

- Czas działania **liniowy** występuje dla algorytmów, w których jest wykonywana pewna stała liczba działań dla każdego z n elementów danych wejściowych.
- Czas działania **liniowo-logarytmiczny** występuje dla algorytmów typu: zadanie rozmiaru n zostaje sprowadzone do dwóch podzadań rozmiaru $n/2$ plus pewna liczba działań, liniowa względem rozmiaru n , potrzebnych najpierw do wykonania rozbicia, a następnie scalenia rozwiązań rozmiaru $n/2$ w rozwiązanie rozmiaru n .

Zadanie: Podaj przykłady takich algorytmów.

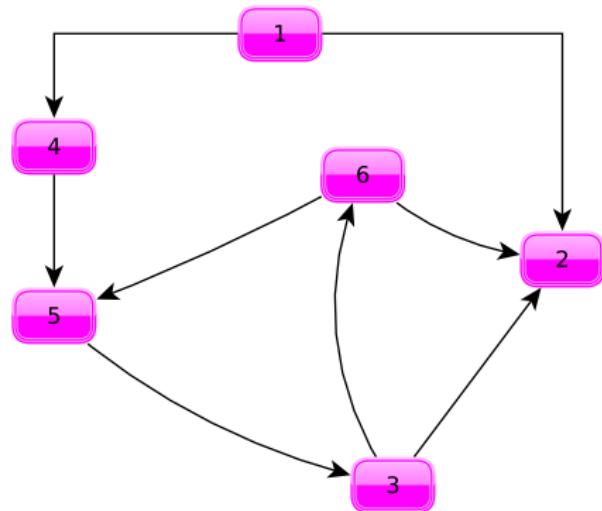
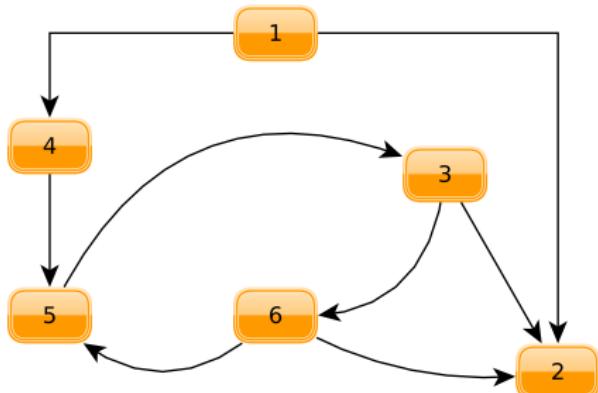
Przykładowe algorytmy i ich złożoności

Czy pokazane dwa grafy są identyczne ?



Przykładowe algorytmy i ich złożoności

Czy pokazane dwa grafy są identyczne ?



Przykładowe algorytmy i ich złożoności

		Węzły					
		1	2	3	4	5	6
Węzły	1		x		x		
	2						
	3		x				x
	4				x		
	5			x			
	6		x			x	

- Problem porównywania grafów z n węzłami (wierzchołkami)

Przykładowe algorytmy i ich złożoności

		Węzły					
		1	2	3	4	5	6
Węzły	1		x		x		
	2						
	3		x				x
	4				x		
	5			x			
	6		x			x	

- Problem porównywania grafów z n węzłami (wierzchołkami)
- Złożoność czasowa: $O(n^2)$

Przykładowe algorytmy i ich złożoności

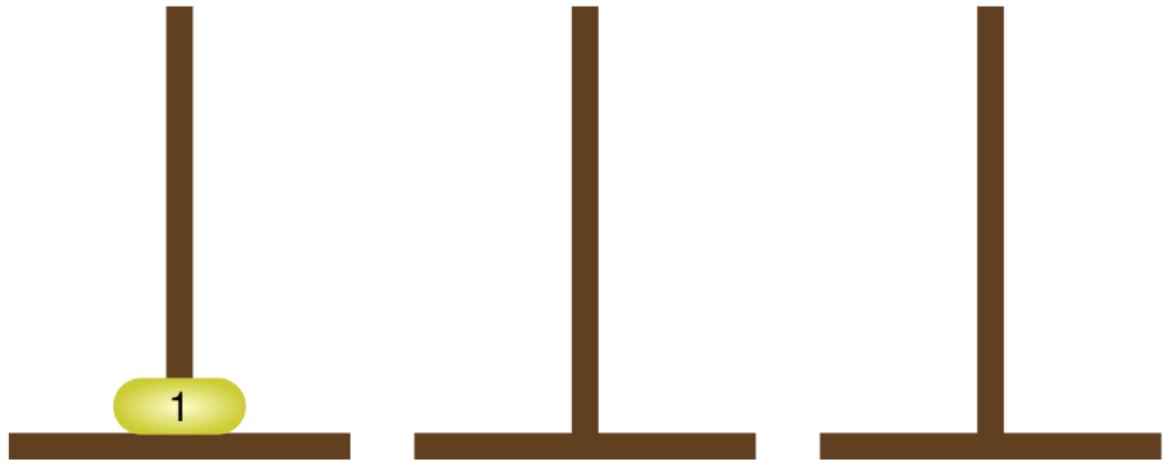
		Węzły					
		1	2	3	4	5	6
W ę z ł y	1		x		x		
	2						
	3		x				x
	4				x		
	5			x			
	6		x			x	

- Problem porównywania grafów z n węzłami (wierzchołkami)
- Złożoność czasowa: $O(n^2)$
- Czas działania **kwadratowy** występuje dla algorytmów, w których jest wykonywana pewna stała liczba działań dla każdej pary elementów.

Zagadka:

Dane sa trzy pale: **A**, **B** i **C**. Na jeden z nich, np. na **A** nałożono krążki o różnych średnicach, tak że krążek mniejszy nie leży pod większym. Pale **B** i **C** sa puste. Należy przenieść wszystkie krążki z pala **A** na **C**. Można używać **B** jako pomocnicze miejsce przechowywania. Nie można kłaść krążków większych na mniejsze.

Wieże Hanoi – 1 Krążek



Wieże Hanoi – 1 Krążek



Dysk przeniesiony z palika 1 na palik 3.

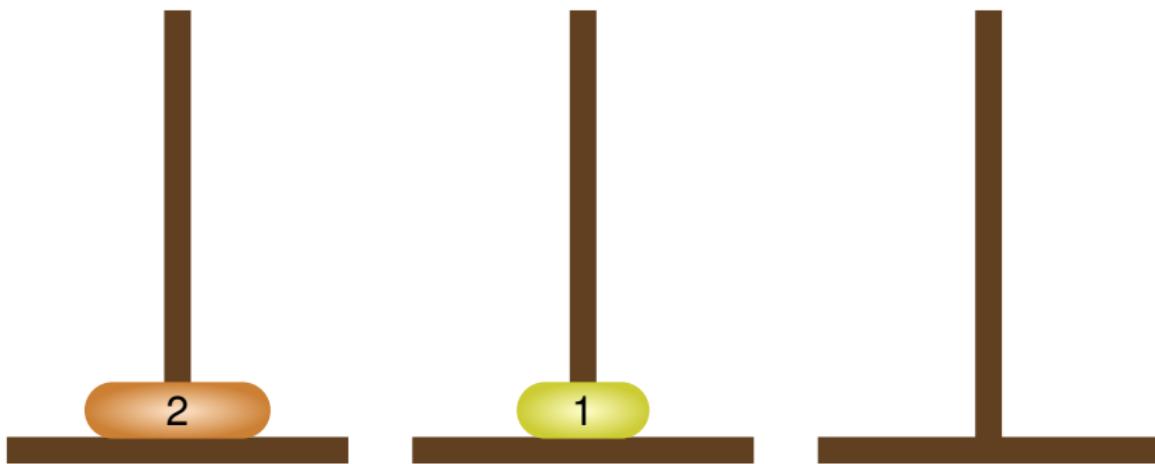
Wieże Hanoi – 1 Krążek



Wieże Hanoi – 2 Krążki

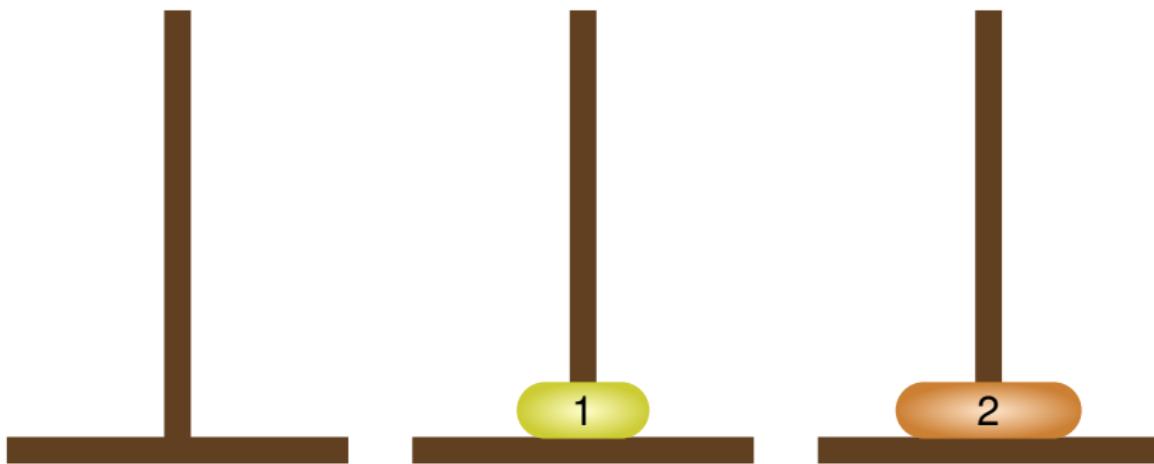


Wieże Hanoi – 2 Krążki



Dysk przeniesiony z palika 1 na palik 2.

Wieże Hanoi – 2 Krążki



Dysk przeniesiony z palika 1 na palik 3.

Wieże Hanoi – 2 Krążki

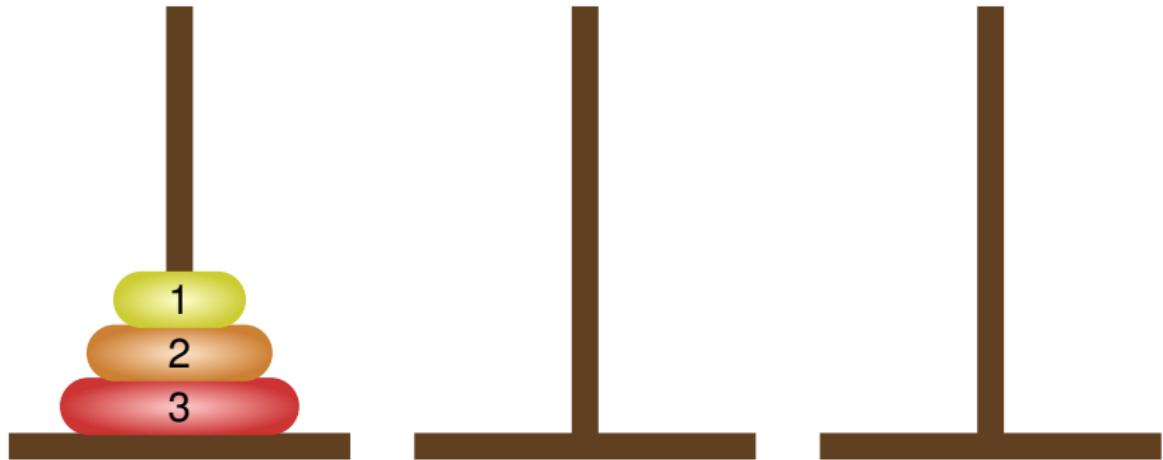


Dysk przeniesiony z palika 2 na palik 3.

Wieże Hanoi – 2 Krążki



Wieże Hanoi – 3 Krążki

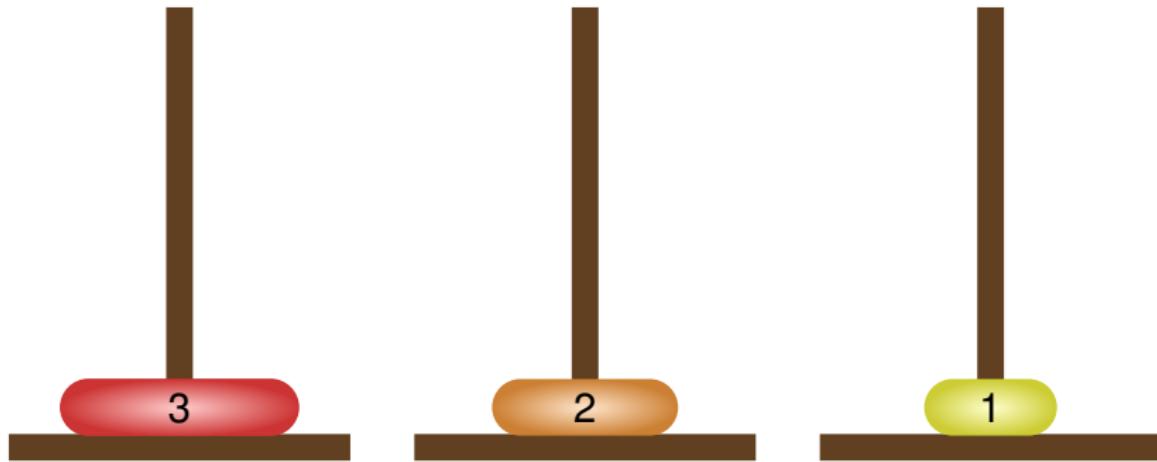


Wieże Hanoi – 3 Krążki



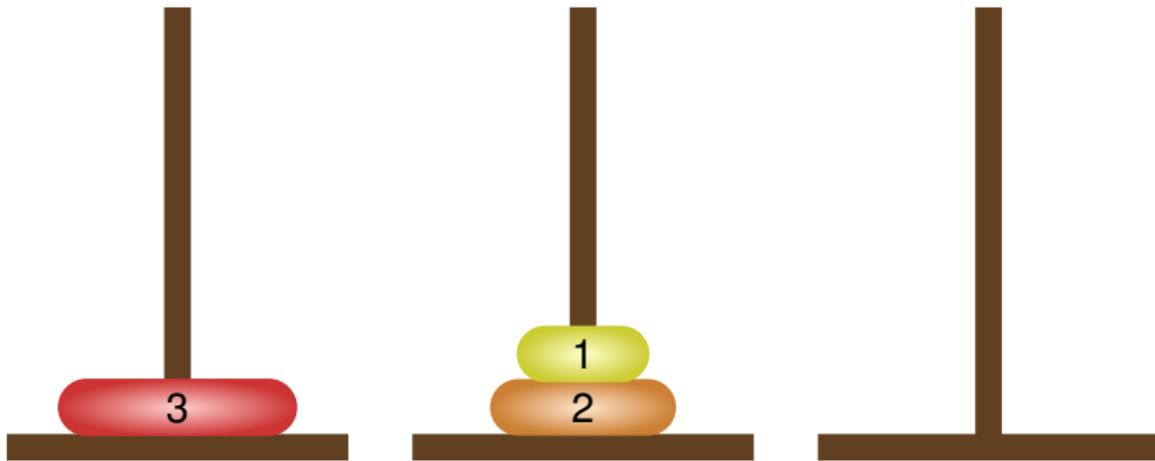
Dysk przeniesiony z palika 1 na palik 3.

Wieże Hanoi – 3 Krążki



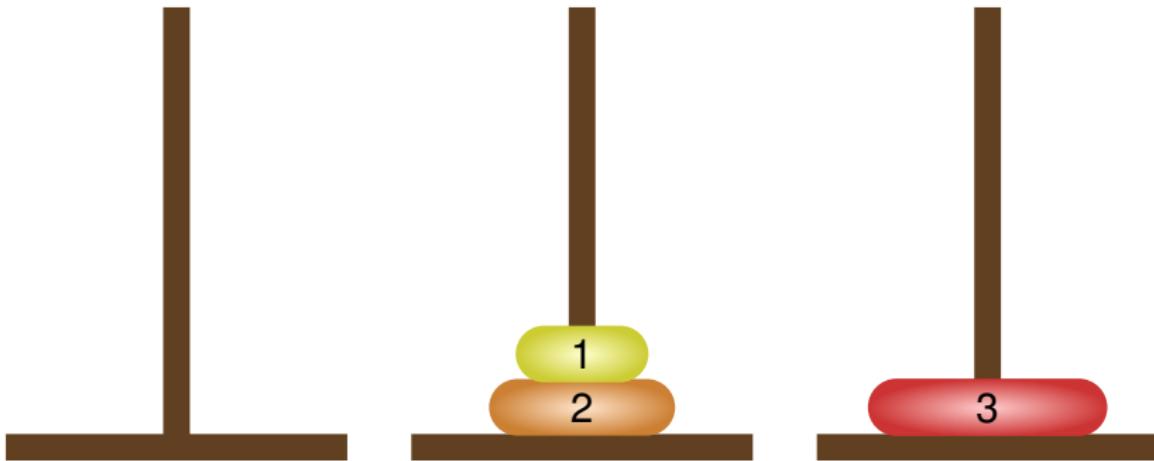
Dysk przeniesiony z palika 1 na palik 2.

Wieże Hanoi – 3 Krążki



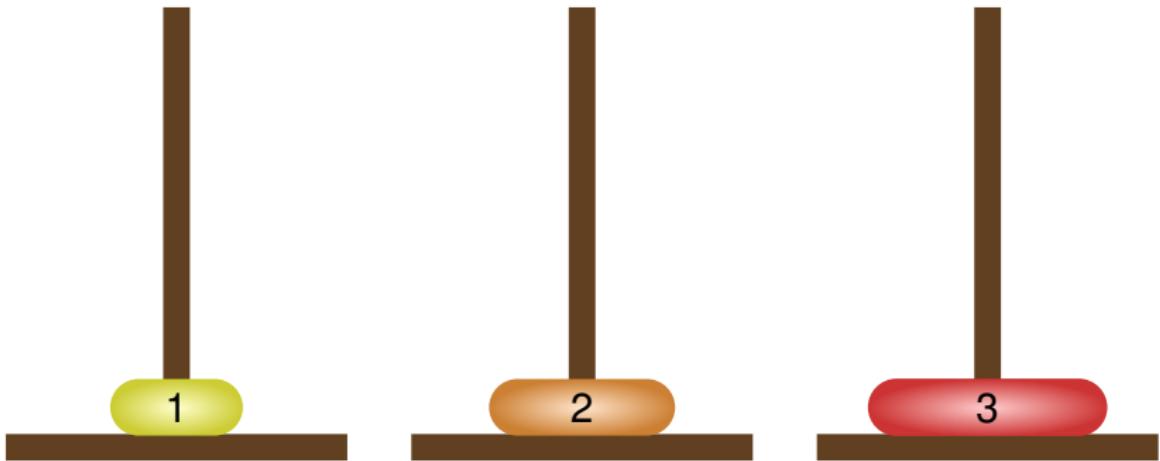
Dysk przeniesiony z palika 3 na palik 2.

Wieże Hanoi – 3 Krążki



Dysk przeniesiony z palika 1 na palik 3.

Wieże Hanoi – 3 Krążki



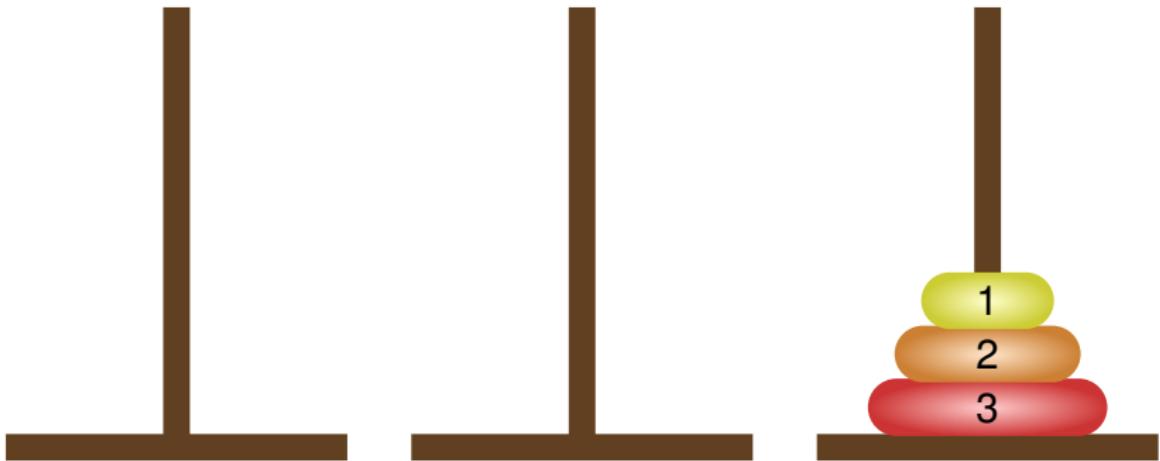
Dysk przeniesiony z palika 2 na palik 1.

Wieże Hanoi – 3 Krążki



Dysk przeniesiony z palika 2 na palik 3.

Wieże Hanoi – 3 Krążki



Dysk przeniesiony z palika 1 na palik 3.

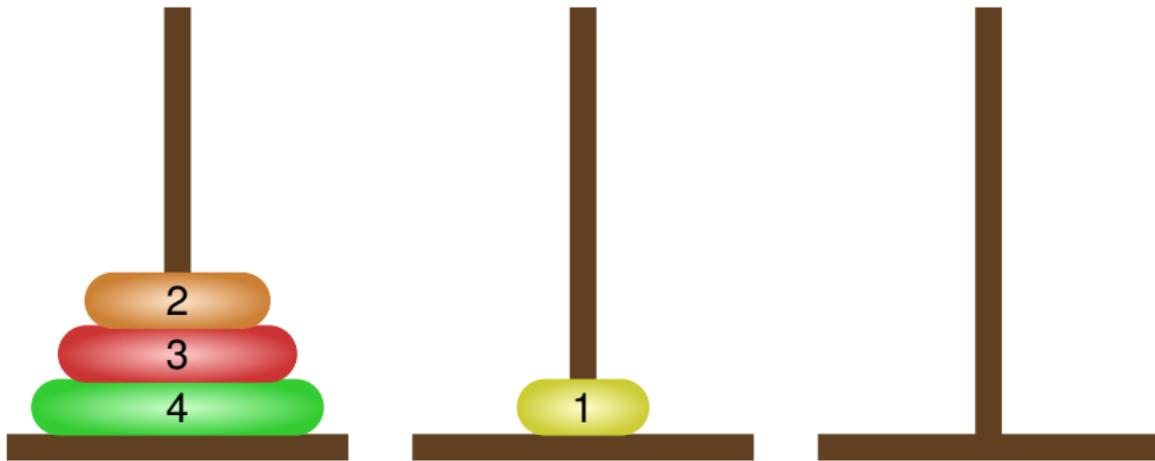
Wieże Hanoi – 3 Krążki



Wieże Hanoi – 4 Krążki

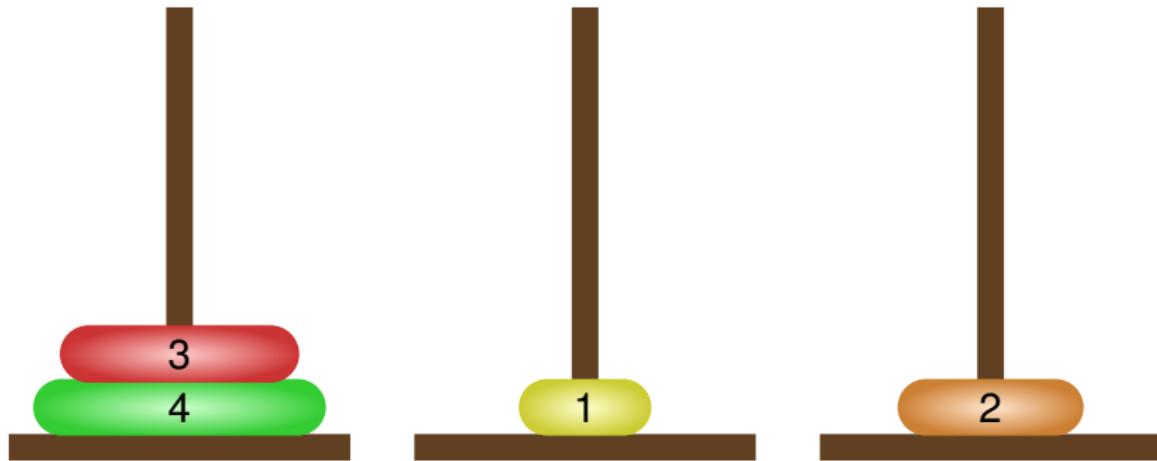


Wieże Hanoi – 4 Krążki



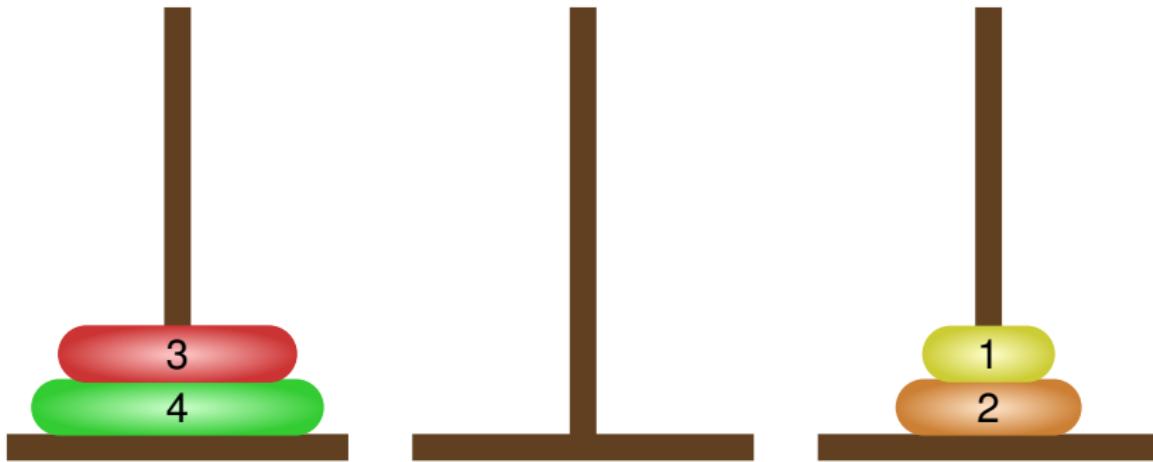
Dysk przeniesiony z palika 1 na palik 2.

Wieże Hanoi – 4 Krążki



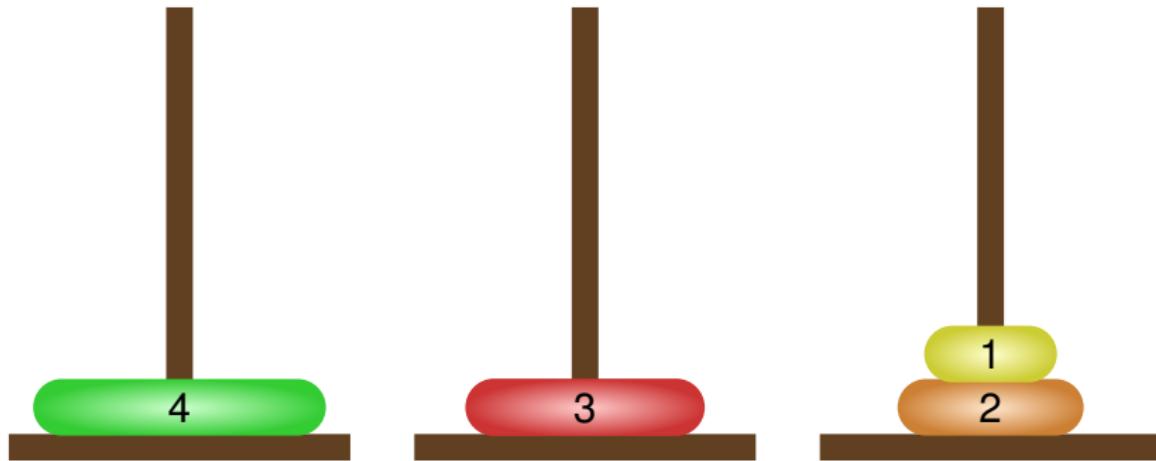
Dysk przeniesiony z palika 1 na palik 3.

Wieże Hanoi – 4 Krążki



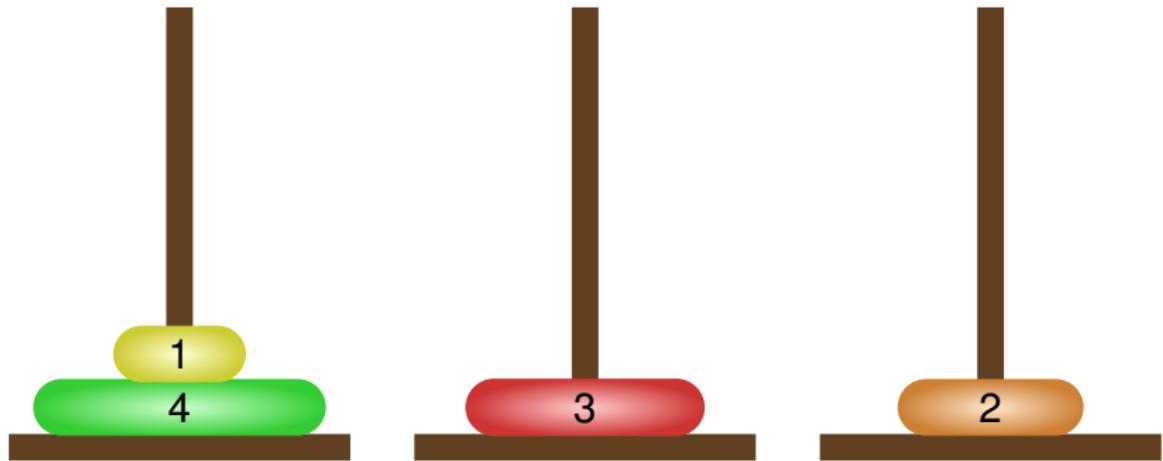
Dysk przeniesiony z palika 2 na palik 3.

Wieże Hanoi – 4 Krążki



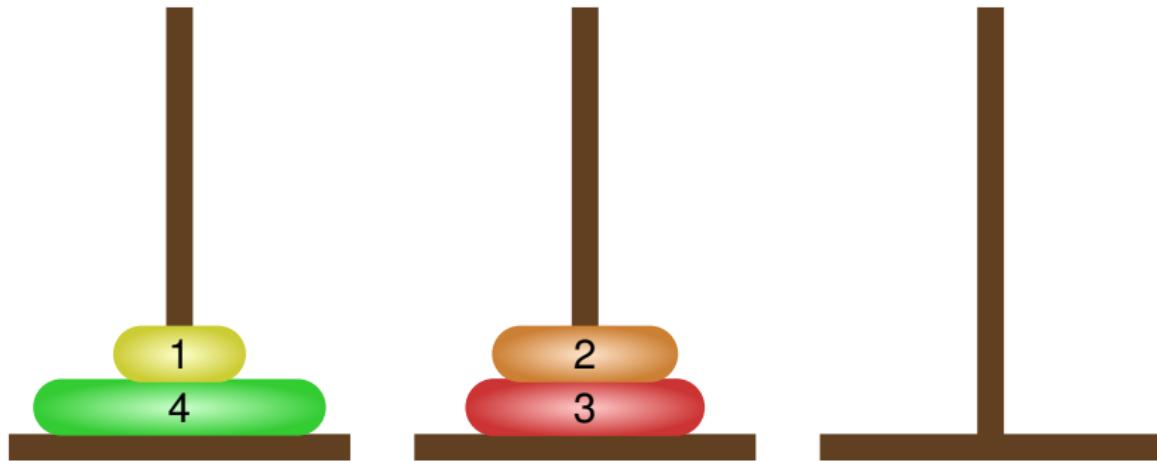
Dysk przeniesiony z palika 1 na palik 2.

Wieże Hanoi – 4 Krążki



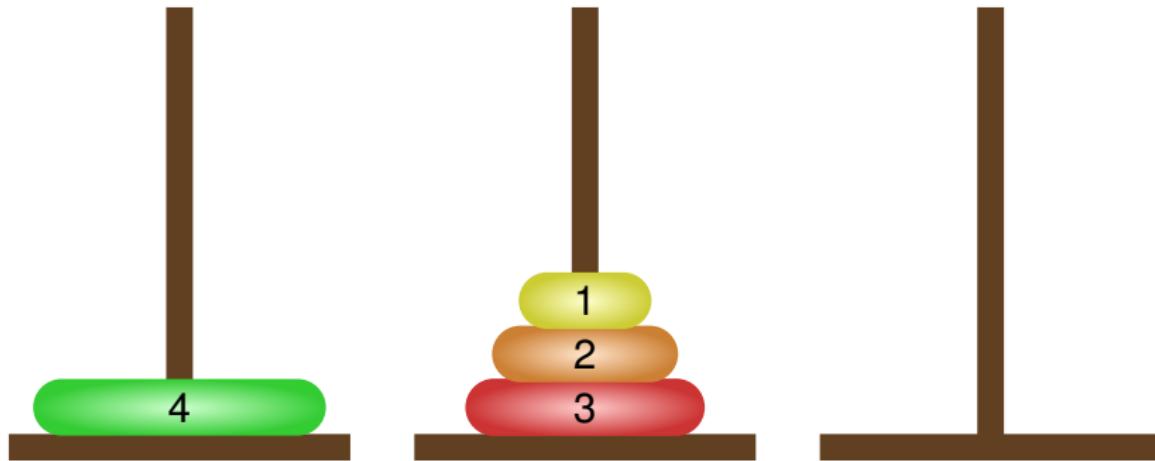
Dysk przeniesiony z palika 3 na palik 1.

Wieże Hanoi – 4 Krążki



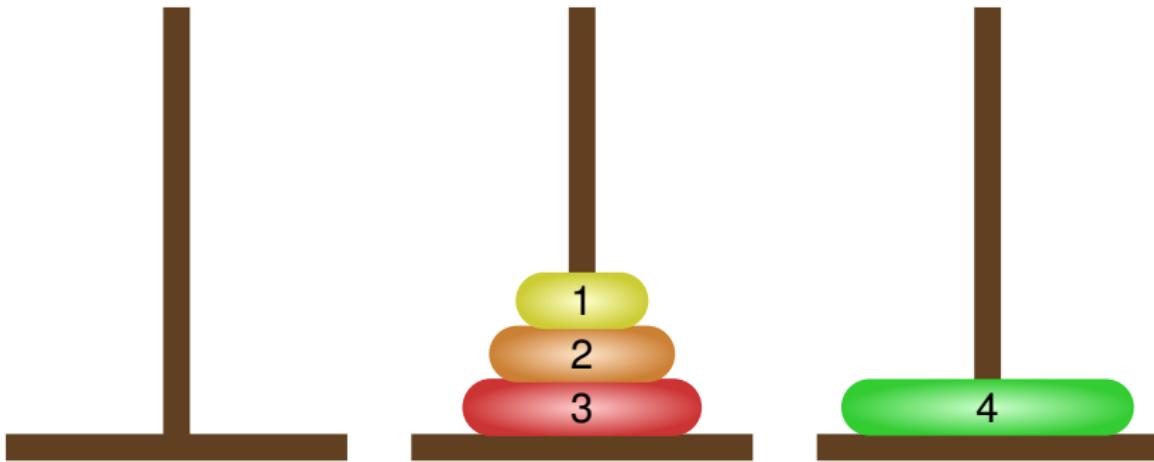
Dysk przeniesiony z palika 3 na palik 2.

Wieże Hanoi – 4 Krążki



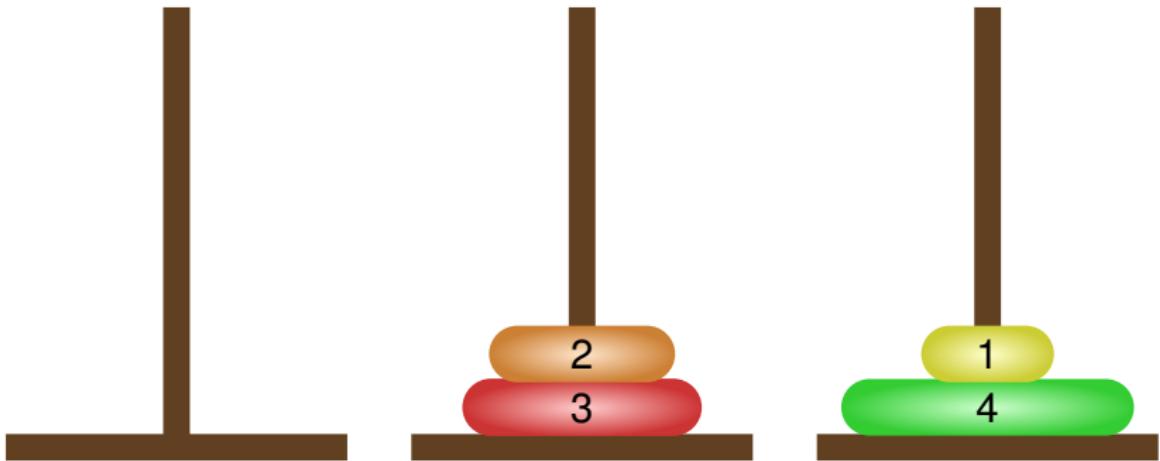
Dysk przeniesiony z palika 1 na palik 2.

Wieże Hanoi – 4 Krążki



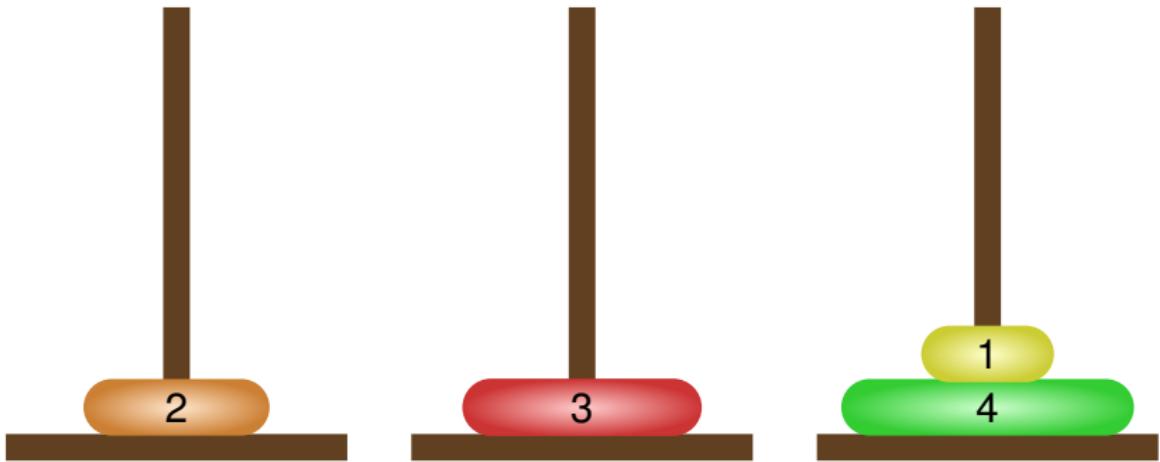
Dysk przeniesiony z palika 1 na palik 3.

Wieże Hanoi – 4 Krążki



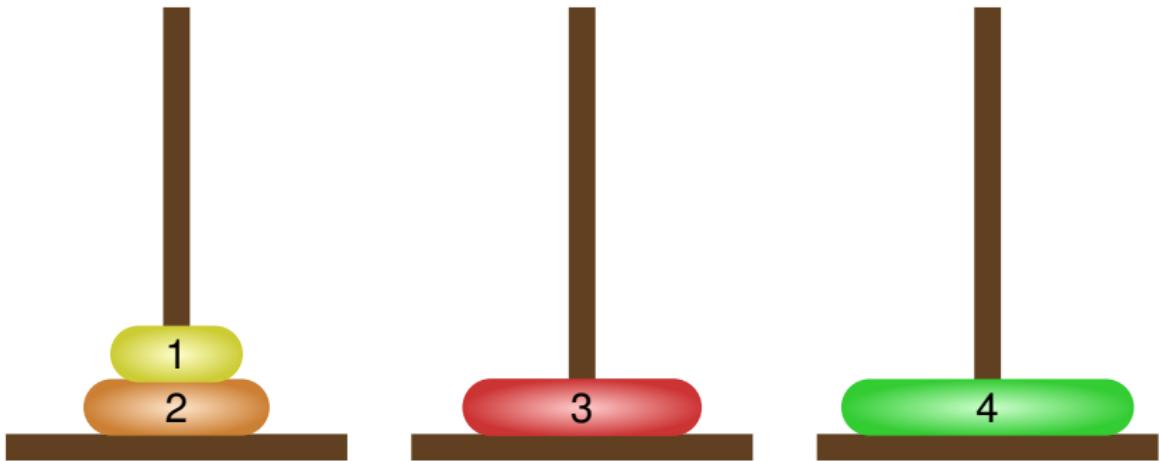
Dysk przeniesiony z palika 2 na palik 3.

Wieże Hanoi – 4 Krążki



Dysk przeniesiony z palika 2 na palik 1.

Wieże Hanoi – 4 Krążki



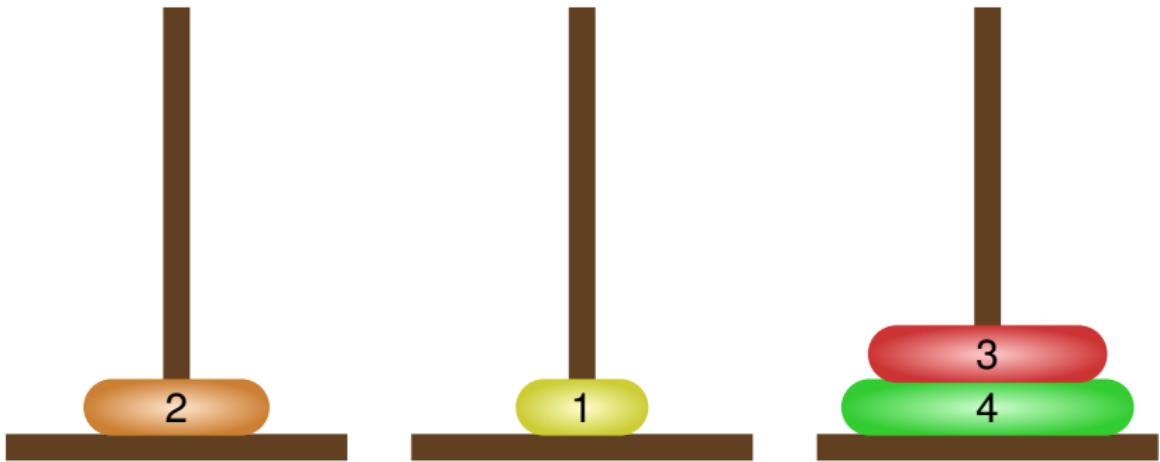
Dysk przeniesiony z palika 3 na palik 1.

Wieże Hanoi – 4 Krążki



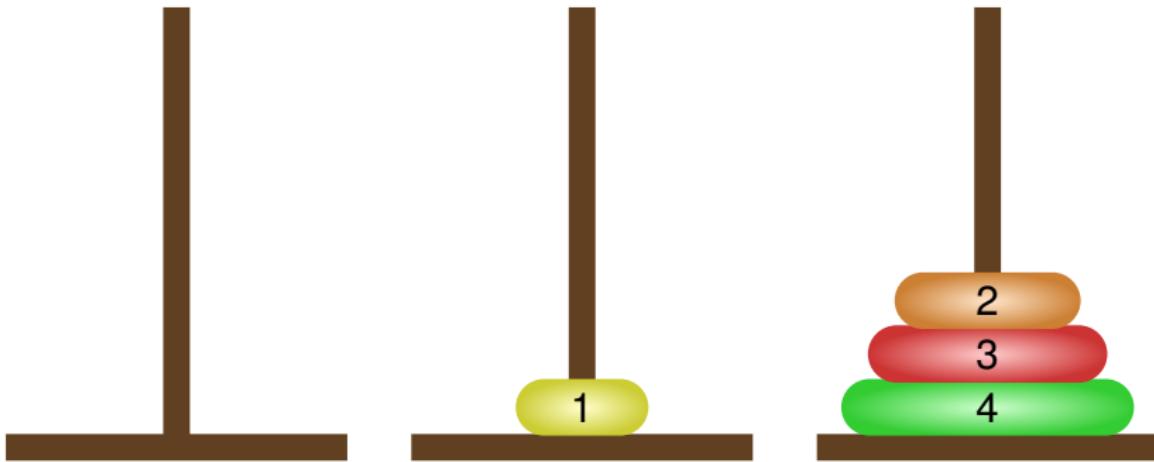
Dysk przeniesiony z palika 2 na palik 3.

Wieże Hanoi – 4 Krążki



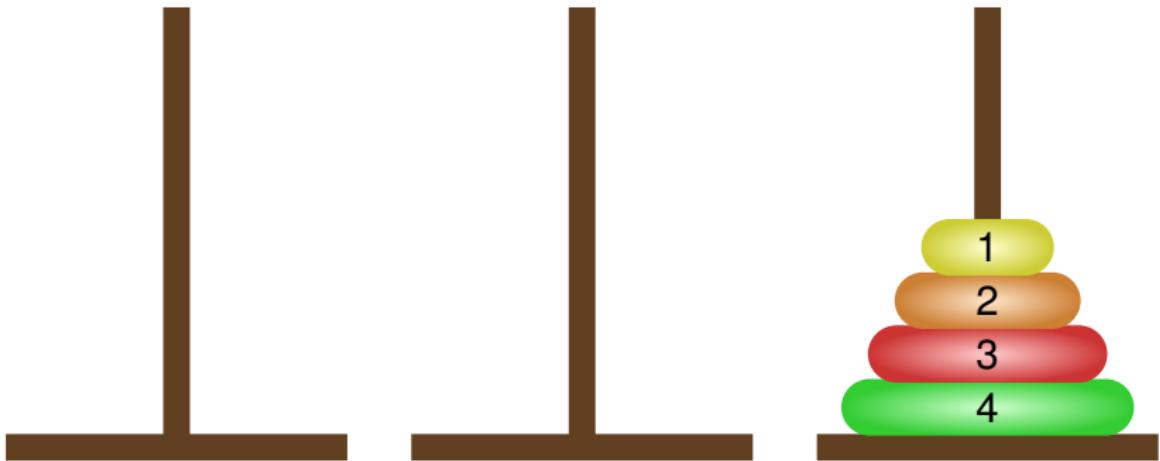
Dysk przeniesiony z palika 1 na palik 2.

Wieże Hanoi – 4 Krążki



Dysk przeniesiony z palika 1 na palik 3.

Wieże Hanoi – 4 Krążki

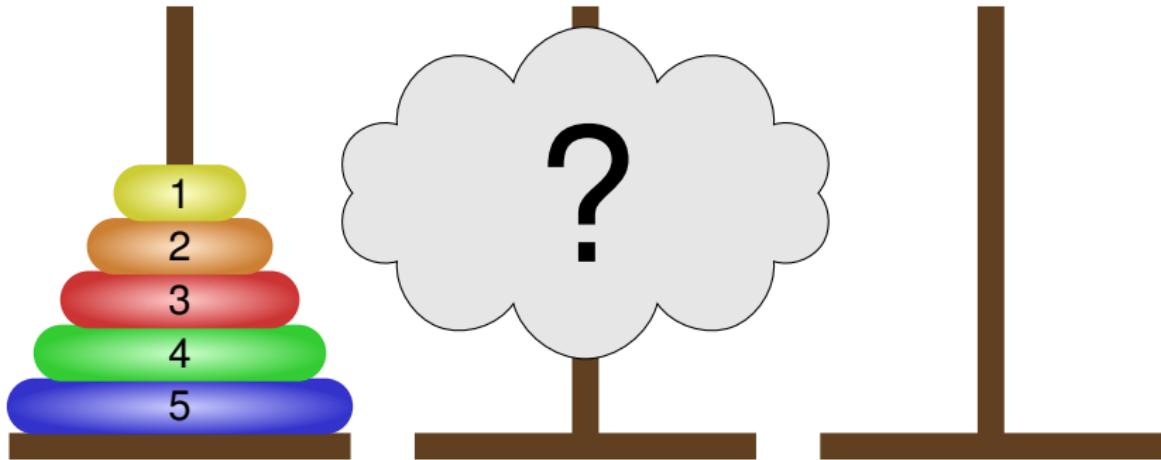


Dysk przeniesiony z palika 2 na palik 3.

Wieże Hanoi – 4 Krążki



Wieże Hanoi – 5 Krążków



Wieże Hanoi

- Zagadka Wież Hanoi stała się znana w XIX wieku dzięki matematykowi **Edouardowi Lucas**, który zaproponował tę zagadkę dla 8 krążków.
- Do sprzedawanego zestawu była dołączona - wymyślona przez Lucas - Tybetańska legenda, według której mnisi w świątyni Brahmy rozwiązują tę łamigłówkę dla 64 złotych krążków.
- Legenda mówi, że gdy mnisi zakończą zadanie, to nastąpi koniec świata.

- Zagadka Wież Hanoi stała się znana w XIX wieku dzięki matematykowi **Edouardowi Lucas**, który zaproponował tę zagadkę dla 8 krążków.
- Do sprzedawanego zestawu była dołączona - wymyślona przez Lucas - Tybetańska legenda, według której mnisi w świątyni Brahmy rozwiązują tę łamigłówkę dla 64 złotych krążków.
- Legenda mówi, że gdy mnisi zakończą zadanie, to nastąpi koniec świata.
- Zakładając, że mnisi wykonują 1 ruch na sekundę, ułożenie wieży zajmie $2^{64} - 1 = 18446744073709551615$ (blisko 18 i pół trylionu) sekund, czyli około 584542 miliardów lat.

Wieże Hanoi

- Zagadka Wież Hanoi stała się znana w XIX wieku dzięki matematykowi **Edouardowi Lucas**, który zaproponował tę zagadkę dla 8 krążków.
- Do sprzedawanego zestawu była dołączona - wymyślona przez Lucas - Tybetańska legenda, według której mnisi w świątyni Brahmy rozwiązują tę łamigłówkę dla 64 złotych krążków.
- Legenda mówi, że gdy mnisi zakończą zadanie, to nastąpi koniec świata.
- Zakładając, że mnisi wykonują 1 ruch na sekundę, ułożenie wieży zajmie $2^{64} - 1 = 18446744073709551615$ (blisko 18 i pół trylionu) sekund, czyli około 584542 miliardów lat.
- Dla porównania: Wszechświat ma około 13,7 mld lat.

Wieże Hanoi

- Zagadka Wież Hanoi stała się znana w XIX wieku dzięki matematykowi **Edouardowi Lucas**, który zaproponował tę zagadkę dla 8 krążków.
- Do sprzedawanego zestawu była dołączona - wymyślona przez Lucas - Tybetańska legenda, według której mnisi w świątyni Brahmy rozwiązują tę łamigłówkę dla 64 złotych krążków.
- Legenda mówi, że gdy mnisi zakończą zadanie, to nastąpi koniec świata.
- Zakładając, że mnisi wykonują 1 ruch na sekundę, ułożenie wieży zajmie $2^{64} - 1 = 18446744073709551615$ (blisko 18 i pół trylionu) sekund, czyli około 584542 miliardów lat.
- Dla porównania: Wszechświat ma około 13,7 mld lat.
- Złożoność czasowa: $O(2^n)$.

Przykładowe algorytmy i ich złożoności

- Czas działania **wykładniczy** ma algorytm, w którym jest wykonywana stała liczba działań dla każdego podzbioru danych wejściowych.

Przykładowe algorytmy i ich złożoności

- Czas działania **wykładniczy** ma algorytm, w którym jest wykonywana stała liczba działań dla każdego podzbioru danych wejściowych.
- Czas działania **siłnie wykładniczy** ma algorytm, w którym jest wykonywana stała liczba działań dla każdej permutacji danych wejściowych.

Przykładowe algorytmy i ich złożoności

- Czas działania **wykładniczy** ma algorytm, w którym jest wykonywana stała liczba działań dla każdego podzbioru danych wejściowych.
- Czas działania **silnie wykładniczy** ma algorytm, w którym jest wykonywana stała liczba działań dla każdej permutacji danych wejściowych.
- Za wewnętrzną cechę algorytmu o złożoności wykładniczej (silnie wykładniczej) uważa się **nierealizowalność**.

Przykładowe algorytmy i ich złożoności

- Czas działania **wykładniczy** ma algorytm, w którym jest wykonywana stała liczba działań dla każdego podzbioru danych wejściowych.
- Czas działania **silnie wykładniczy** ma algorytm, w którym jest wykonywana stała liczba działań dla każdej permutacji danych wejściowych.
- Za wewnętrzną cechę algorytmu o złożoności wykładniczej (silnie wykładniczej) uważa się **nierealizowalność**.

Zadanie: Podaj przykłady takich algorytmów.

Porównanie czasów realizacji algorytmu wykładowicznego na dwóch komputerach

Rozmiar n	20	50	100	200
Czas działania $2^n/10^6$	1,04 sec.	35,7 lat	$4 \cdot 10^{14}$ wieków	$5 \cdot 10^{44}$ wieków
Czas działania $2^n/10^9$	0,001 sec.	13 dni	$4 \cdot 10^{11}$ wieków	$5 \cdot 10^{41}$ wieków

Grafy A

na podstawie wykładu prof. dr hab. Elżbieta Richter-Wąs z UJ

Graf to jest relacja binarna.

Dla grafów mamy ogromne możliwości wizualizacji jako zbiór punktów (zwanych wierzchołkami) połączonych liniami lub strzałkami (nazwanych krawędziami). Pod tym względem graf stanowi uogólnienie drzewiastego modelu danych. Podobnie jak drzewa, grafy występują w różnych postaciach: grafów skierowanych i nieskierowanych lub etykietowanych i nieetykietowanych.

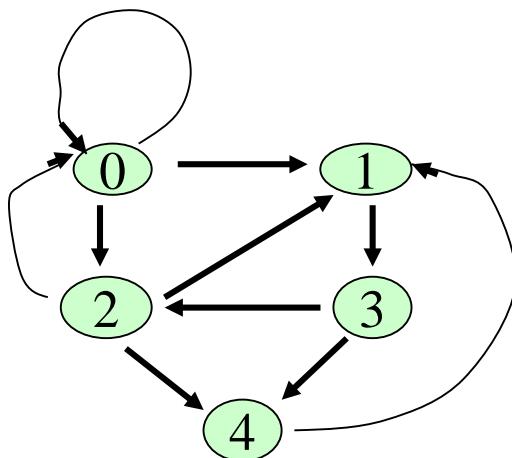
Grafy są przydatne do analizy szerokiego zakresu problemów: obliczanie odległości, znajdowanie cykliczności w relacjach, reprezentacji struktury programów, reprezentacji relacji binarnych, reprezentacji automatów i układów elektronicznych.

Podstawowe pojęcia

Graf skierowany (ang. directed graph)

Składa się z następujących elementów:

- (1) Zbioru N wierzchołków (ang. nodes)
- (2) Relacji binarnej A na zbiorze N . Relacje A nazywa się zbiorem krawędzi (ang. arcs) grafu skierowanego.
Krawędzie stanowią zatem pary wierzchołków (u,v) .

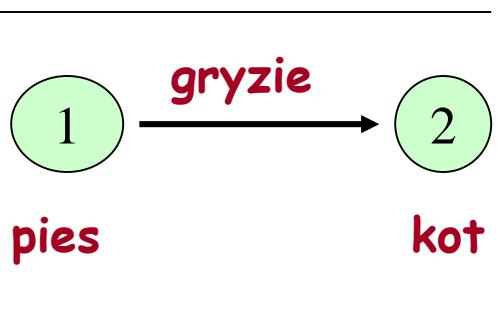


$$N = \{0, 1, 2, 3, 4\}$$

$$A = \{ (0,0), (0,1), (0,2), (1,3), (2,0), (2,1), (2,4), (3,2), (3,4), (4,1) \}$$

Etykiety

Podobnie jak dla drzew, dla grafów istnieje możliwość przypisania do każdego wierzchołka etykiety (ang. label).



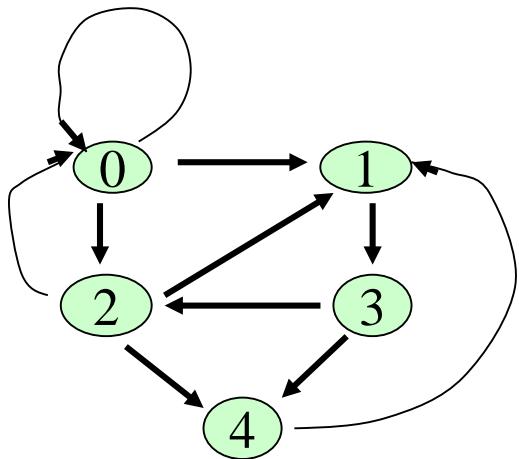
Nie należy mylić nazwy wierzchołka z jego etykietą. Nazwy wierzchołków muszą być niepowtarzalne, ale kilka wierzchołków może być oznaczonych ta sama etykieta.

Drogi

Droga (ang. path) w grafie skierowanym stanowi listę wierzchołków, (v_1, v_2, \dots, v_k) taka, że występuje krawędź łącząca każdy wierzchołek z następnym, to znaczy $v_i \rightarrow v_{i+1}$ dla $i=1,2,\dots,k$. Długość (ang. length) drogi wynosi $k-1$, co stanowi liczbę krawędzi należących do tej samej drogi.

Grafy cykliczne i acykliczne

Cykl (ang. cycle) w grafie skierowanym jest drogą o długości 1 lub więcej, która zaczyna się i kończy w tym samym wierzchołku. Długość cyklu jest długością drogi. Cykl jest prosty (ang. simple) jeżeli żaden wierzchołek (oprócz pierwszego) nie pojawia się więcej niż raz.



Cykle proste:

(0,0), (0,2,0), (1,3,2,1), (1,3,2,4,1)

Cykl nieprosty:

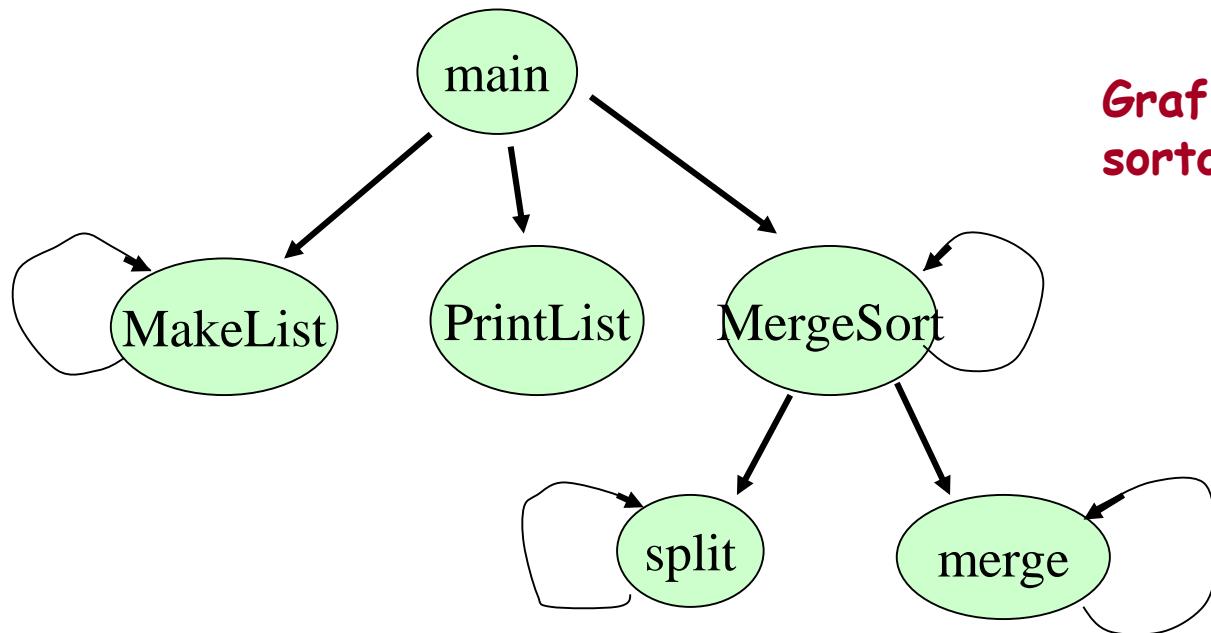
(0,2,1,3,2,0)

Jeżeli istnieje cykl nieprosty zawierający wierzchołek v , to można znaleźć prosty cykl który zawiera v . Jeżeli graf posiada jeden lub więcej cykli to mówimy że jest grafem cyklicznym (ang. cyclic). Jeżeli cykle nie występują to, graf określa się mianem acyklicznego (ang. acyclic).

Grafy wywołań

Wywołania dokonywane przez zestaw funkcji można reprezentować za pomocą grafu skierowanego, zwanego grafem wywołań. Jego wierzchołki stanowią funkcje, a krawędź $P \rightarrow Q$ istnieje wówczas, gdy funkcja P wywołuje funkcję Q . Istnienie cyklu w grafie implikuje występowanie w algorytmie rekurencji.

Rekurencja w której funkcja wywołuje samą siebie nazywamy bezpośrednią (ang. direct). Czasem mamy do czynienia z rekurencją pośrednią (ang. indirect) która reprezentuje cykl o długości większej niż 1. np. $P \rightarrow Q \rightarrow R \rightarrow P$

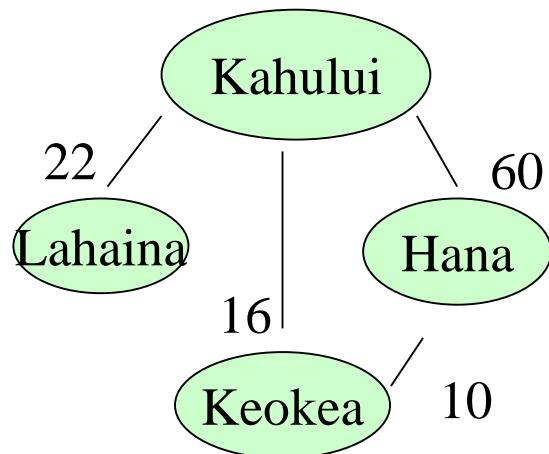


Graf wywołań dla algorytmu sortowania przez scalanie

rekurencja
bezpośrednia

Grafy nieskierowane

Czasem zasadne jest połączenie wierzchołków krawędziami, które nie posiadają zaznaczonego kierunku. Z formalnego punktu widzenia taka krawędź jest zbiorem dwóch wierzchołków. Zapis $\{u,v\}$ mówi ze wierzchołki u oraz v są połączone w dwóch kierunkach. Jeśli $\{u,v\}$ jest krawędzią nieskierowaną, wierzchołki u i v określa się jako sąsiednie (ang. adjacent) lub mianem sąsiadów (ang. neighbors). Graf zawierający krawędzie nieskierowane, czyli graf z relacją symetryczności krawędzi, nosi nazwę grafu nieskierowanego (ang. undirected graph).



Graf nieskierowany reprezentujący drogi na wyspie Hwajow Maui.

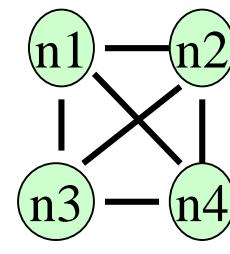
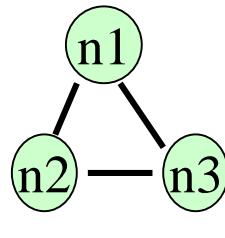
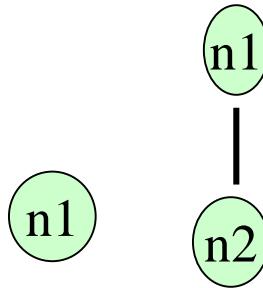
Droga to lista wierzchołków. Nieco trudniej jest sprecyzować co to jest cykl, tak aby nie była to każda lista $(v_1, v_2, \dots, v_{k-1}, v_k, v_{k-1}, \dots, v_2, v_1)$

Wybrane pojęcia z teorii grafów

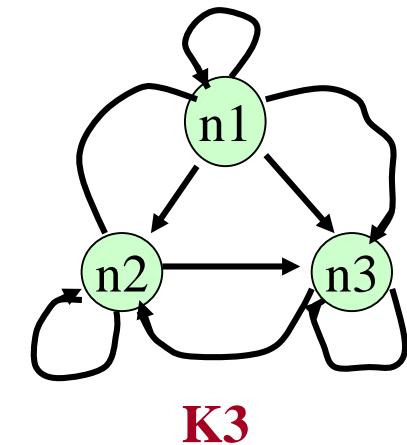
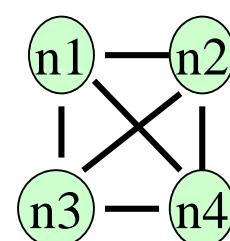
Teoria grafów jest dziedziną matematyki zajmującą się właściwościami grafów.

Grafy pełne

Nieskierowany graf posiadający krawędzie pomiędzy każdą parą różnych wierzchołków nosi nazwę grafu pełnego (ang. complete graph). Graf pełny o n wierzchołkach oznacza się przez K_n .



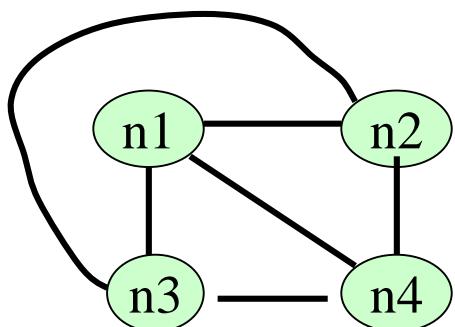
K_4



Liczba krawędzi w nieskierowanym grafie K_n wynosi $n(n-1)/2$, w skierowanym grafie K_n wynosi n^2 .

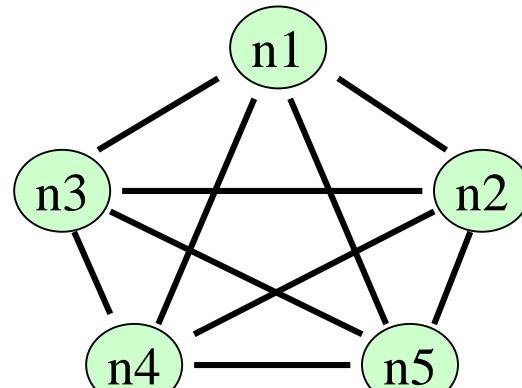
Grafy planarne i nieplanarne

O grafie nieskierowanym mówi się, że jest planarny (ang. planar) wówczas, gdy istnieje możliwość rozmieszczenia jego wierzchołków na płaszczyźnie, a następnie narysowania jego krawędzi jako linii ciągły, które się nie przecinają. Grafy nieplanarne (ang. nonplanar) to takie, które nie posiadają reprezentacji płaskiej.

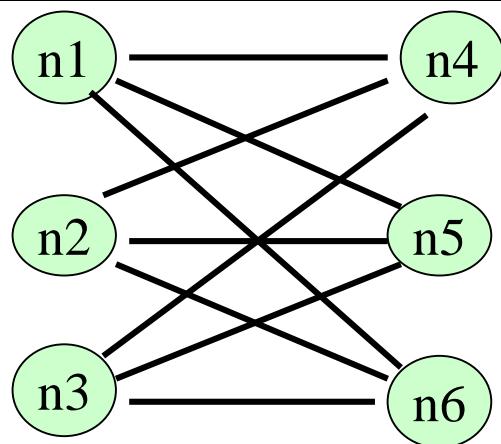


K4

reprezentacja planarna



K5



K3,3

najprostsze grafy nieplanarne

Zastosowania planarnosci i kolorowanie grafów

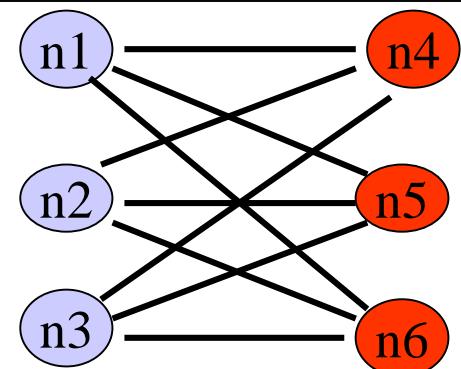
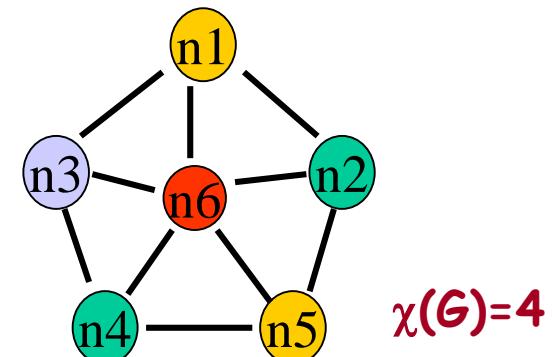
Planarnosc ma duże zastosowanie do graficznych reprezentacji w informatyce dla projektowania różnego rodzaju układów (np. scalonych, bramek, etc.)

Kolorowanie grafu (ang. graph coloring) polega na przypisaniu do każdego wierzchołka pewnego koloru, tak aby żadne dwa wierzchołki połączone krawędzią nie miały tego samego koloru.

Minimalna liczba kolorów potrzebna do takiej operacji nazwana jest liczbą chromatyczną grafu (ang. chromatic number), oznaczaną $\chi(G)$.

(1) Jeżeli graf jest pełny to jego liczba chromatyczna jest równą liczbie wierzchołków

(2) Jeżeli graf możemy pokolorować przy pomocy dwóch kolorów to nazywamy go dwudzielnym (ang. bipartite graph). Np. $K_{3,3}$.



Sposoby implementacji grafów

Istnieją dwie standardowe metody reprezentacji grafów. Pierwsza z nich, listy sąsiedztwa (ang. adjacency lists), jest, ogólnie rzecz biorąc, podobna do implementacji relacji binarnych. Druga, macierze sąsiedztwa (ang. adjacency matrices), to nowy sposób reprezentowania relacji binarnych, który jest bardziej odpowiedni dla relacji w przypadku których liczba istniejących par stanowi znaczącą część całkowitej liczby par, jakie mogłyby teoretycznie istnieć w danej dziedzinie.

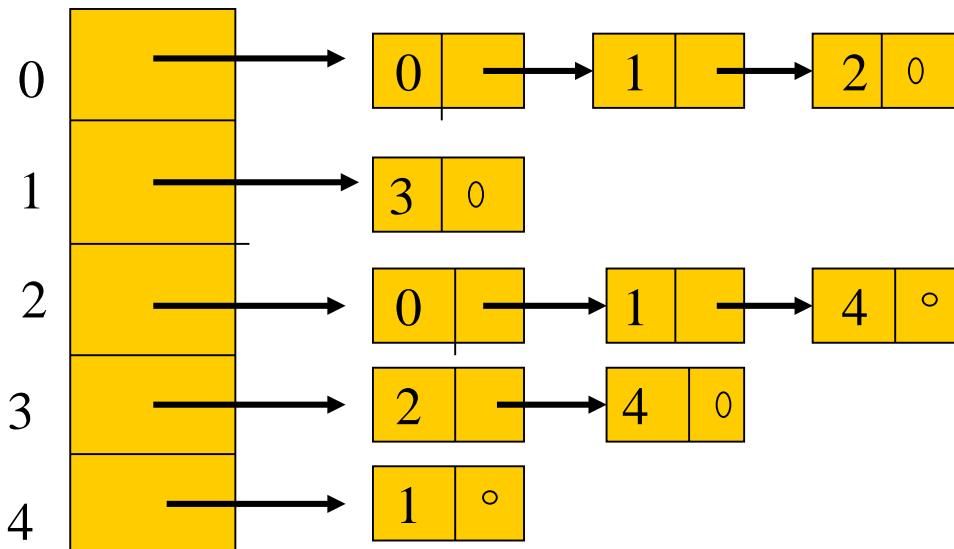
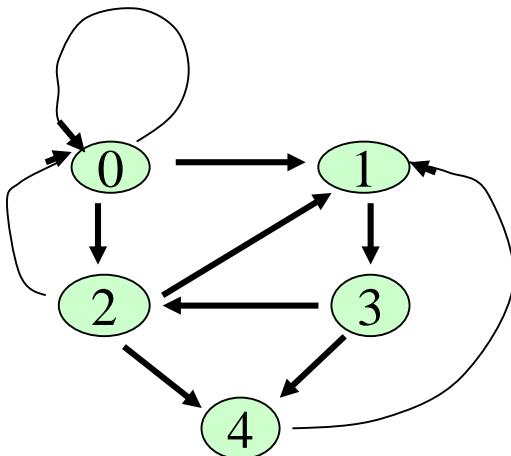
Listy sąsiedztwa

```
typedef struct CELL *LIST;  
struct CELL {  
    NODE nodeName;  
    LIST next;  
}  
LIST successors[MAX]
```

Wierzchołki są ponumerowane kolejnymi liczbami całkowitymi 0,1,....., MAX-1 lub oznaczone za pomocą innego adekwatnego typu wyliczeniowego (używamy poniżej typu NODE jako synonimy typu wyliczeniowego). Wówczas można skorzystać z podejścia opartego na wektorze własnym.

Element successors[u] zawiera wskaźnik do listy jednokierunkowej wszystkich bezpośrednich następców wierzchołka u. Następcy mogą występować w dowolnej kolejności na liście jednokierunkowej.

Reprezentacja grafu za pomocą list sąsiedztwa



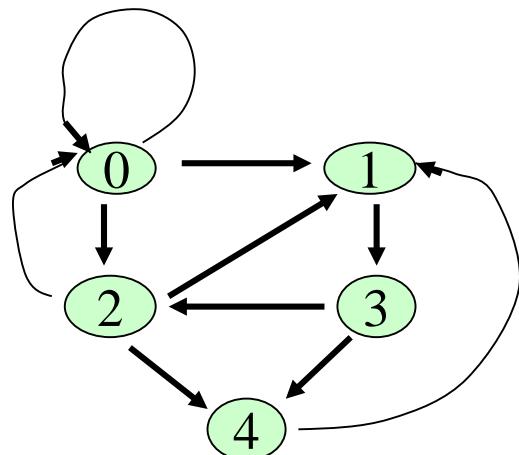
Listy sąsiedztwa zostały posortowane wg. kolejności, ale następcy mogą występować w dowolnej kolejności na odpowiedniej liście sąsiedztwa.

Reprezentacja grafu za pomocą macierzy sąsiedztwa

Tworzymy dwuwymiarową tablicę:

BOOLEAN arcs [MAX][MAX];

w której element $\text{arcs}[u][v]$ ma wartość TRUE wówczas, gdy istnieje krawędź $u \rightarrow v$, zaś FALSE, gdy taka krawędź nie istnieje.



	0	1	2	3	4
0	1	1	1	0	0
1	0	0	0	1	0
2	1	1	0	0	1
3	0	0	1	0	1
4	0	1	0	0	0

Porównanie macierzy sąsiedztwa z listami sąsiedztwa.

Macierze sąsiedztwa są preferowanym sposobem reprezentacji grafów wówczas, gdy grafy są gęste (ang. dense), to znaczy, kiedy liczba krawędzi jest bliska maksymalnej możliwej ich liczby. Dla grafy skierowanego o n wierzchołkach maksymalna liczba krawędzi wynosi n^2 . Jeśli graf jest rzadki (ang. sparse) to reprezentacja oparta na listach sąsiedztwa może pozwolić zaoszczędzić pamięć.

Istotne różnice między przedstawionymi reprezentacjami grafów są widoczne już przy wykonywaniu prostych operacji.

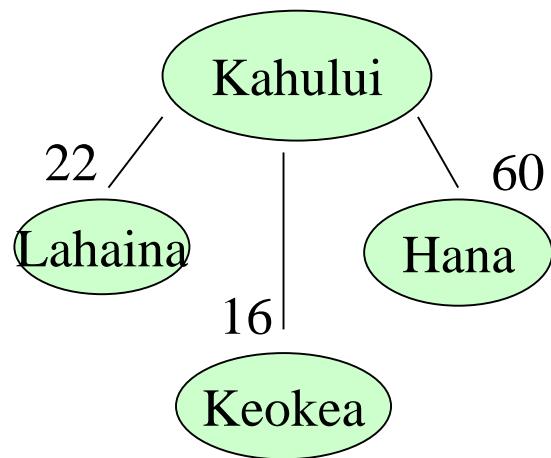
Preferowany sposób reprezentacji:

OPERACJA	GRAF GESTY	GRAF RZADKI
Wyszukiwanie krawędzi	Macierz sąsiedztwa	Obie
Znajdowanie następców	Obie	Lista sąsiedztwa
Znajdowanie poprzedników	Macierz sąsiedztwa	Obie

Spójna składowa grafu nieskierowanego

Każdy graf nieskierowany można podzielić na jedną lub większą liczbę spójnych składowych (ang. connected components).

Każda spójna składowa to taki zbiór wierzchołków, że dla każdych dwóch z tych wierzchołków istnieje łącząca je ścieżka. Jeżeli graf składa się z jednej spójnej składowej to mówimy ze jest spójny (ang. connected).



to jest graf spójny

Spójne składowe jako klasy równoważności

Pojęcia spójnych składowych można potraktować formalnie jako klasy równoważności relacji równoważności P zdefiniowanej na wierzchołkach grafu nieskierowanego jako uPv wtedy i tylko wtedy, gdy istnieje droga z wierzchołka u do v .

Sprawdzenie że P jest relacją równoważności

(1) Relacja P jest zwrotna, to znaczy zachodzi uPu dla dowolnego wierzchołka u , gdyż istnieje droga o długości 0 z dowolnego wierzchołka do niego samego.

(2) Relacja P jest symetryczna. Jeśli zachodzi uPv , to istnieje droga z wierzchołka u do v . Ponieważ graf jest nieskierowany, odwrotny porządek wierzchołków również stanowi drogę. Stąd zachodzi vPu .

(3) Relacja P jest przechodnia. Założymy, że relacje uPW oraz wPv są prawdziwe. Wówczas istnieje pewna droga, na przykład

(x_1, x_2, \dots, x_j)

z u do w . zatem $u=x_1$ oraz $w=x_j$. Ponadto istnieje droga (y_1, y_2, \dots, y_k) z wierzchołka w do v , gdzie $w=y_1$ oraz $v=y_k$. Składając obie drogi razem otrzymujemy drogę z u do v , czyli

$(u=x_1, x_2, \dots, x_j=w=y_1, y_2, \dots, y_k=v)$

Relacja P dzieli graf na klasy równoważności. Każda klasa równoważności zdefiniowana relacją drogi odpowiada spójnej składowej tego grafu.

Algorytm wyznaczania spójnych składowych

Chcemy określić spójne składowe grafu G . Przeprowadzamy rozumowanie indukcyjne.

Podstawa:

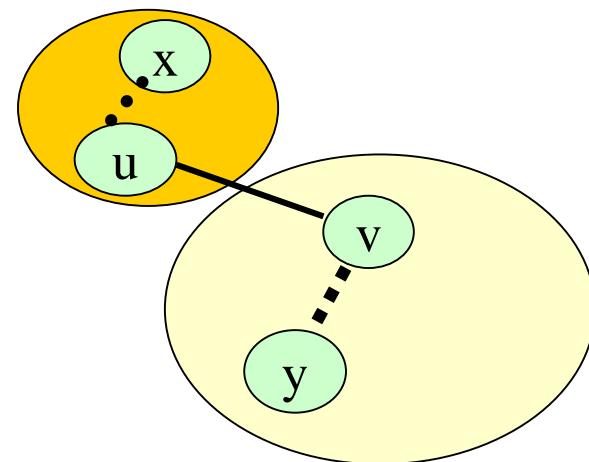
Graf G_0 zawiera jedyni wierzchołki grafu G i żadnej jego krawędzi. Każdy wierzchołek stanowi odrębną spójną składową.

Indukcja:

Zakładamy, że znamy już spójne składowe grafu G_i po rozpatrzeniu pierwszych i krawędzi, a obecnie rozpatrujemy $(i+1)$ krawędź $\{u,v\}$.

(a) jeżeli krawędź $\{u,v\}$ należy do jednej spójnej składowej to nic się nie zmienia

(b) jeżeli do dwóch różnych, to łączymy te dwie spójne składowe w jedną.



Struktura danych dla wyznaczania spójnych składowych

Biorąc pod uwagę przedstawiony algorytm, musimy zapewnić szybką wykonywalność następujących operacji.

(1) gdy jest określony wierzchołek to znajdź jego bieżącą spójną składową

(2) połącz dwie spójne składowe w jedną

Zaskakująco dobre wyniki daje ustawienie wierzchołków każdej składowej w strukturze drzewiastej, gdzie spójna składowa jest reprezentowana przez korzeń.

(a) aby wykonać operacje (1) należy przejść do korzenia

(b) aby wykonać operacje (2) wystarczy korzeń jednej składowej określić jako potomka korzenia składowej drugiej. Przyjmiemy zasadę ze korzeń drzewa o mniejszej wysokości czynimy potomkiem.

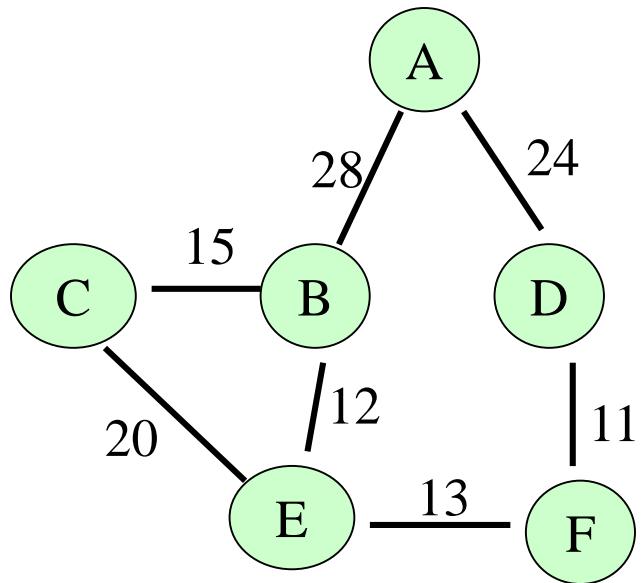
Przy takiej konstrukcji czas wykonania instrukcji (1) jest $O(\log(n))$, czas wykonania instrukcji (2) jest $O(1)$. Wyznaczenie wszystkich spójnych składowych to $O(m \log(n))$ gdzie m =liczba krawędzi, n =liczba wierzchołków.

Minimalne drzewa rozpinające

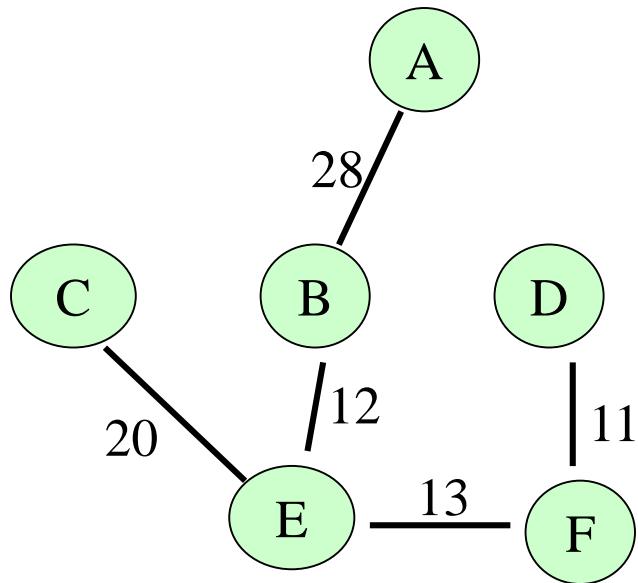
Drzewo rozpinające (ang. spanning tree) grafu nieskierowanego G stanowi zbiór wierzchołków tego grafu wraz z podzbiorem jego krawędzi, takich że:

- (1) Łączą one wszystkie wierzchołki, czyli istnieje droga między dwoma dowolnymi wierzchołkami która składa się tylko z krawędzi drzewa rozpinającego.
- (2) Tworzą one drzewo nie posiadające korzenia, nieuporządkowane. Oznacza to że nie istnieją żadne (proste) cykle.

Jeśli graf G stanowi pojedynczą spójną składową to drzewo rozpinające zawsze istnieje. Minimalne drzewo rozpinające (ang. minimal spanning tree) to drzewo rozpinające, w którym suma etykiet jego krawędzi jest najmniejsza ze wszystkich możliwych do utworzenia drzew rozpinających tego grafu.



Graf nieskierowany



Drzewo rozpinające

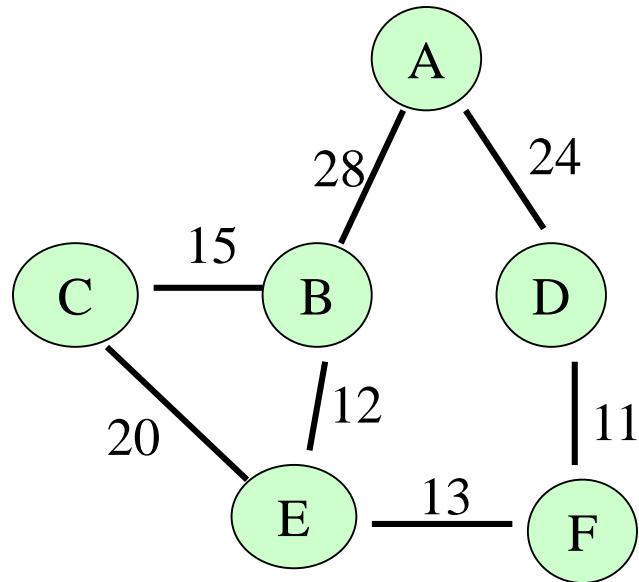
Znajdowanie minimalnego drzewa rozpinającego

Istnieje wiele algorytmów. Jeden z nich to algorytm Kruskala, który stanowi proste rozszerzenie algorytmu znajdowania spójnych składowych.

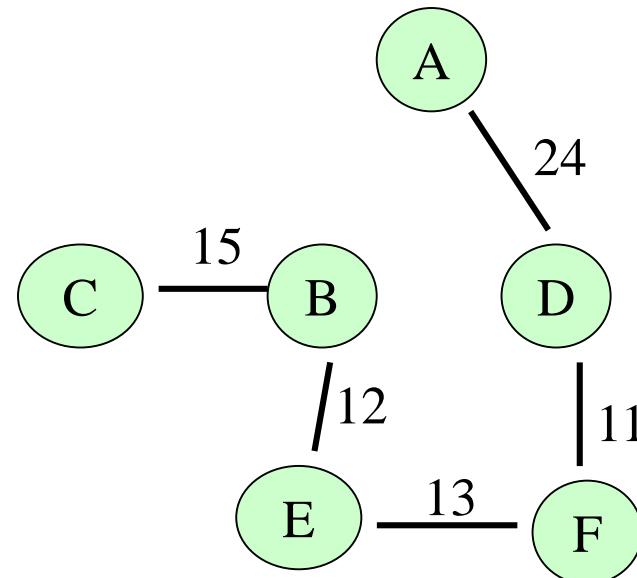
Wymagane zmiany to:

(1) należy rozpatrywać krawędzie w kolejności zgodnej z rosnącą wartością ich etykiet.

(2) należy dołączyć krawędź do drzewa rozpinającego tylko w takim wypadku gdy jej końce należą do dwóch różnych spójnych składowych



Graf nieskierowany



Minimalne drzewo rozpinające

Algorytm Kruskala jest dobrym przykładem algorytmu zachłanego (ang. greedy algorithm), w przypadku którego podejmowany jest szereg decyzji, z których każdą stanowi wybranie opcji najlepszej w danym momencie. Lokalnie podejmowane decyzje polegają w tym przypadku na wyborze krawędzi dodawanej do formowanego drzewa rozpinającego.

Z każdym razem wybierana jest krawędź o najmniejsze wartości etykiety, która nie narusza definicji drzewa rozpinającego, zabraniającej utworzenia cyklu.

Dla algorytmu Kruskala można wykazać, że jego rezultat jest optymalny globalnie, to znaczy że daje on w wyniku drzewo rozpinające o minimalnej wadze.

Czas wykonania algorytmu jest $O(m \log n)$ gdzie n to jest liczba wierzchołków , a m to jest większa z wartości liczby wierzchołków i liczby krawędzi.

Uzasadnienie poprawności algorytmu Kruskala

Niech G będzie nieskierowanym grafem spójnym.

(Dla niektórych etykiet dopusczamy dodanie nieskończenie malej wartości tak aby wszystkie etykiety były różne. graf G będzie miał wobec tego unikatowe minimalne drzewo rozpinające, które będzie jednym spośród minimalnych drzew rozpinających grafu G o oryginalnych wagach)

Niech ciąg e_1, e_2, \dots, e_m oznacza wszystkie krawędzie grafu G w kolejności zgodnej z rosnącą wartością ich etykiet, rozpoczynając od najmniejszej.

Niech K będzie drzewem rozpinającym grafu G o odpowiednio zmodyfikowanych etykietach, utworzonym przez zastosowanie algorytmu Kruskala, a T niech będzie unikatowym minimalnym drzewem rozpinającym grafu G .

Należy udowodnić że K i T stanowią to samo drzewo. Jeśli są różne musi istnieć co najmniej jedna krawędź, która należy do jednego z nich a nie należy do drugiego.

Uzasadnienie poprawności algorytmu Kruskala

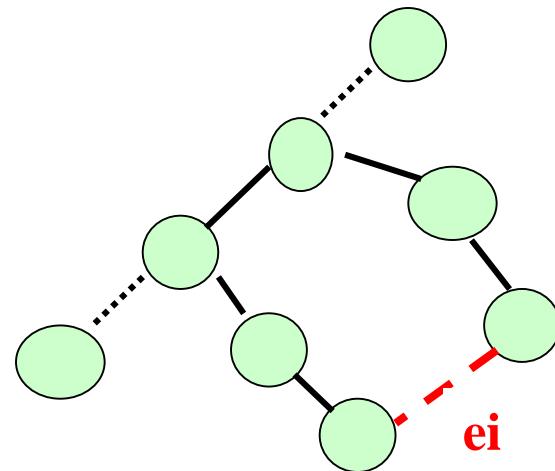
Niech e_i oznacza pierwszą taka krawędź spośród uporządkowanych krawędzi, to znaczy każda z krawędzi e_1, e_2, \dots, e_{i-1} należy do obu drzew K i T albo nie należy do żadnego z nich. Istnieją dwa przypadki w zależności czy krawędź e_i należy do drzewa K czy do drzewa T . W każdym z tych przypadków wykażemy sprzeczność, co będzie stanowić dowód, że e_i nie może istnieć, a stąd że $K=T$, oraz że K stanowi minimalne drzewo rozpinające grafu G .

Przypadek 1

krawędź e_i należy do T , ale nie należy do K .

Jeżeli algorytm Kruskala odrzuca e_i , oznacza to że e_i formuje cykl z pewna droga P , utworzona z uprzednio wybranych krawędzi drzewa K .

Jeżeli krawędzie drogi P należą do K to należą także do T . A więc $P + e_i$ utworzyłaby cykl w T co jest sprzeczne z definicja drzewa rozpinającego. Stąd niemożliwe jest aby e_i należała do T a nie należała do K .



Droga P (linia ciągła) należy zarówno do drzewa T jak i K ; krawędź ei należy tylko do T

Przypadek 2

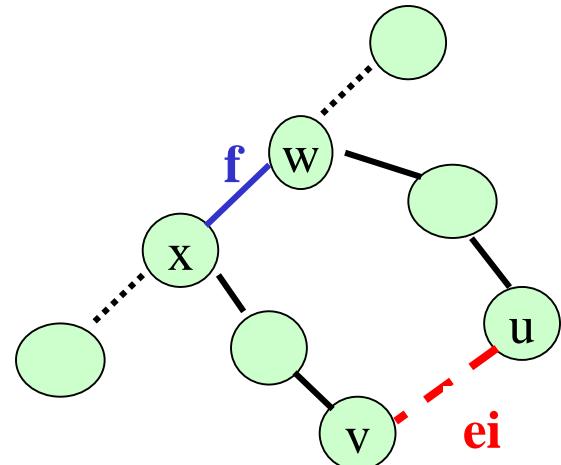
krawędź ei należy do K, ale nie należy do T.

Niech krawędź ei łączy wierzchołki u i v.

Ponieważ drzewo T jest spójne, musi istnieć w T pewna acykliczna droga z wierzchołka u do v. Niech nosi ona nazwę Q. Ponieważ w skład Q nie wchodzi ei, Q + ei tworzy cykl prosty w grafie G.

(a) krawędź ei posiada najwyższą wartość etykiety. Musiałoby to oznaczać ze K zawiera cykl co jest niemożliwe.

(b) na drodze Q istnieje krawędź f która ma wartość etykiety wyższą niż ei. Można by wiec usunąć f a wprowadzić ei nie niszcząc spójności. A wiec rozpięte drzewo miałoby wartość mniejszą niż wartość dla T co jest w sprzeczności z początkowym twierdzeniem ze T jest minimalne.



**Droga Q (linia ciągła)
należy do drzewa T;
można dodać
krawędź ei i usunąć
krawędź f**

Algorytmy i Struktury Danych.

Grafy

dr hab. Bożena Woźna-Szcześniak
bwozna@gmail.com

Jan Długosz University, Poland

Wykład 7



Grafy - Obszary Zastosowania

- Informatyka
- Technika ¹
- Fizyka
- Nauki społeczne
- Biologia
- Kartografia
- Lingwistyka
- i wiele innych ...

¹ Deo N. Teoria grafów i jej zastosowania w technice i informatyce, PWN, Warszawa 1980.

Grafy - Zastosowania w Informatyce:

- Reprezentacja struktury programów
- Modelowanie systemów komputerowych
- Automatyczna weryfikacja systemów współbieżnych i rozproszonych
- Kryptografia - Szyfr Cezara, Problem Dalekopisu, Kod Graya, Kody Huffmana, kryptografia wizualna, itd.
- Sieci komputerowe - znajdowanie najkrótszych ścieżek
- i wiele innych ...

Graf - idea

- **Graf** to - intuicyjnie - zbiór wierzchołków, które mogą być połączone krawędziami, w taki sposób, że każda krawędź kończy się i zaczyna w pewnym wierzchołku.
- Wierzchołki grafu zazwyczaj są etykietowane (numerowane) i reprezentują pewne obiekty (np. miasta).
- Krawędzie obrazują relacje pomiędzy obiektami (np. połączenia kolejowe).
- Krawędzie mogą mieć wyznaczony kierunek, a graf zawierający takie krawędzie nazywany jest grafem skierowanym.
- Krawędź może posiadać wagę, tzn. przypisaną wartość liczbową, która określa na przykład odległość w kilometrach pomiędzy wierzchołkami (jeśli graf jest reprezentacją połączeń między miastami).

Graf - Leonharda Eulera

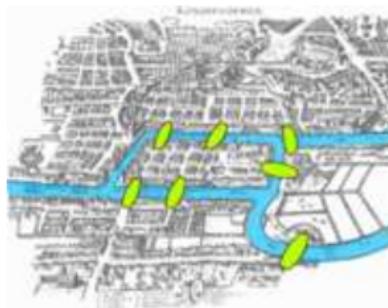
- Za pierwszego teoretyka i badacza grafów uważa się *Leonharda Euler²*, który rozstrzygnął tzw. zagadnienie mostów królewieckich.



²Leonhard Euler (ur. 15 kwietnia 1707 r. w Bazylei - Szwajcaria, zm. 18 września 1783 r. w Petersburgu - Rosja) - szwajcarski matematyk, fizyk i astronom, jeden z twórców nowoczesnej matematyki.

Mosty królewieckie

Przez Królewiec przepływała rzeka Pregole, w której rozwidleniach znajdowały się dwie wyspy. Ponad rzeką przerzucono siedem mostów, z których jeden łączył obie wyspy, a pozostałe mosty łączyły wyspy z brzegami rzeki. Plan mostów pokazuje rysunek:

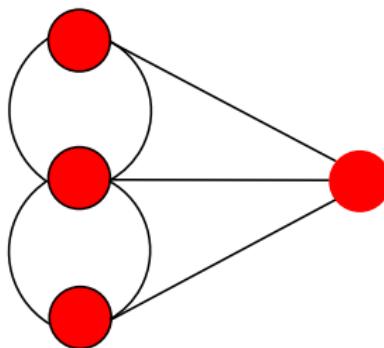


Mosty królewieckie

- Zwykłe spacerowanie szybko się znudziło mieszkańców Królewca i zaczęli zastanawiać się, czy istnieje taka trasa spacerowa, która przechodzi przez każdy most dokładnie raz, żadnego nie omija, i pozwala wrócić do punktu wyjścia.
- Mieszkańcy nie potrafili rozwiązać postawionego problemu samodzielnie, więc postanowili napisać do matematyka *Leonharda Eulera*.

Mosty królewieckie

- Euler wykazał, że rozwiązanie problemu mieszkańców nie jest możliwe, a decyduje o tym nieparzysta liczba wylotów mostów zarówno na każdą z wysp, jak i na oba brzegi rzeki. (Jeśli wejdzie się po raz trzeci na wyspę, nie ma jak z niej wyjść).
- Sytuację tę można przedstawić za pomocą następującego grafu:



Mosty królewieckie

- Problem mostów królewieckich, to inaczej problem znalezienia w danym grafie tzw. **cyklu Eulera**, czyli cyklu przechodzącego przez wszystkie wierzchołki i wszystkie krawędzie danego grafu, ale przez każdą krawędź tylko raz.
- W roku 1736 roku Euler udowdnił twierdzenie, które obecnie można sformułować w sposób następujący:

W grafie można znaleźć cykl Eulera wtedy i tylko wtedy, gdy graf jest spójny i każdy jego wierzchołek ma parzysty stopień.

Graf skierowany

Definicja

Grafem skierowanym (digrafem) nazywamy strukturę

$$G = (V, E)$$

gdzie

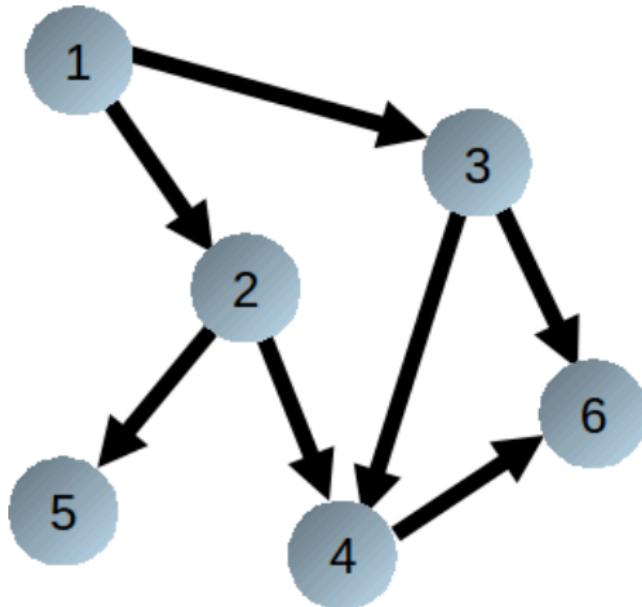
- V to zbiór wierzchołków,
- $E \subseteq \{(u, v) : u, v \in V\}$ to zbiór uporządkowanych par wierzchołków ze zbioru V , zwanych krawędziami.

Przez n oznaczamy ilość wierzchołków (rozmiar zbioru V)

Przez m oznaczamy ilość krawędzi (rozmiar zbioru E)

Graf skierowany $G = (V, E)$ - przykład

- $V = \{1, 2, 3, 4, 5, 6\}$
- $E = \{(1, 2), (1, 3), (2, 4), (2, 5), (3, 4), (3, 6), (4, 6)\}$
- $n = 6, m = 7$



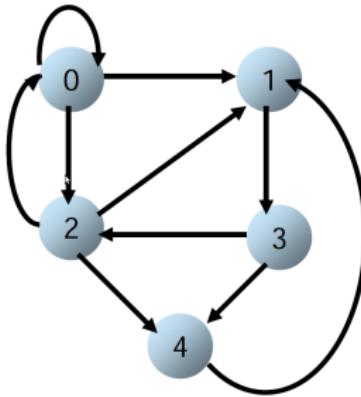
- **Droga (ścieżka)** w grafie skierowanym to lista wierzchołków (n_1, n_2, \dots, n_k) taka, że występuje krawędź łącząca każdy wierzchołek z następnym, to znaczy $(n_i, n_{i+1}) \in E$ dla $i = 1, 2, \dots, k$.
- **Droga prosta**, to droga w której żadna krawędź się nie powtarza.
- **Długość drogi** wynosi $k - 1$, co stanowi liczbę krawędzi należących do tej samej drogi.
- **Cykl** w grafie skierowanym jest drogą o długości co najmniej 1, która zaczyna się i kończy w tym samym wierzchołku. Długość cyklu jest długością drogi.
- **Cykl jest prosty**, jeżeli żaden wierzchołek (oprócz pierwszego) nie pojawia się w nim więcej niż raz.

Podstawowe Pojęcia

- Jeżeli graf posiada jeden lub więcej cykli to mówimy, że jest **grafem cyklicznym**.
- Jeżeli graf nie posiada cykli to, mówimy, że jest **grafem acyklicznym**.

Podstawowe Pojęcia

- Jeżeli graf posiada jeden lub więcej cykli to mówimy, że jest **grafem cyklicznym**.
- Jeżeli graf nie posiada cykli to, mówimy, że jest **grafem acyklicznym**.
- Przykład:



Cykle proste: $(0, 0), (0, 2, 0), (1, 3, 2, 1), (1, 3, 2, 4, 1)$

Cykl nieprosty: $(0, 2, 1, 3, 2, 0)$

Graf nieskierowany

Definicja

Grafem nieskierowanym (grafem) nazywamy strukturę

$$G = (V, E)$$

gdzie

- V to zbiór wierzchołków,
- $E \subseteq \{\{u, v\} : u, v \in V\}$ to zbiór dwuelementowych podzbiorów/multizbiorów zbioru V , zwanych krawędziami.

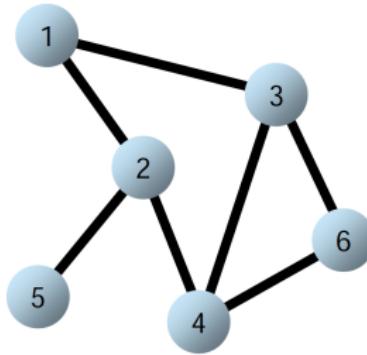
Przez n oznaczamy ilość wierzchołków (rozmiar zbioru V)

Przez m oznaczamy ilość krawędzi (rozmiar zbioru E)

Droga w grafie nieskierowanym, to lista wierzchołków.

Graf nieskierowany - przykład

- $V = \{1, 2, 3, 4, 5, 6\}$
- $E = \{(1, 2), (1, 3), (2, 4), (2, 5), (3, 4), (3, 6), (4, 6)\}$
- $n = 6, m = 7$



Podstawowe pojęcia

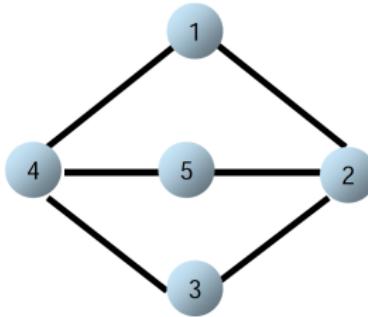
- Graf posiadający krawędzie pomiędzy każdą parą różnych wierzchołków nosi nazwę grafu **pełnego**.
- O grafie nieskierowanym mówi się że jest **planarny**, jeśli istnieje możliwość rozmieszczenia jego wierzchołków na płaszczyźnie, a następnie narysowania jego krawędzi jako linii ciągłych które się nie przecinają.
- Grafy **nieplanarne** to takie, które nie posiadają reprezentacji płaskiej.

Podstawowe pojęcia

- Graf posiadający krawędzie pomiędzy każdą parą różnych wierzchołków nosi nazwę grafu **pełnego**.
- O grafie nieskierowanym mówi się że jest **planarny**, jeśli istnieje możliwość rozmieszczenia jego wierzchołków na płaszczyźnie, a następnie narysowania jego krawędzi jako linii ciągłych które się nie przecinają.
- Grafy **nieplanarne** to takie, które nie posiadają reprezentacji płaskiej.
- Planarność ma duże zastosowanie w informatyce, m.in., w graficznej reprezentacji różnego rodzaju układów (np. scalonych, bramek, etc.).

- **Stopień wierzchołka:**
 - W grafie nieskierowanym to liczba incydentnych z nim krawędzi.
 - W grafie skierowanym to suma stopni wejściowego (ilość kończących się krawędzi) i wyjściowego (ilość wychodzących krawędzi).
- **Graf regularny** to graf, w którym każdy wierzchołek ma taki sam stopień.
- **f -graf** to graf z ograniczonym stopniem wierzchołka, tzn. jego stopień nie może być większy niż f .

Przykład



- Stopień wierzchołka 1 jest równy 2, a wierzchołka 2 jest równy 3.
- Graf jest planarny.
- 3-graf.

Reprezentacja grafów w komputerze

- Macierz sąsiedztwa
- Lista incydencji (sąsiedztwa)
- Lista krawędzi
- Macierz incydencji

Macierz sąsiedztwa

Dany jest graf $G = (V, E)$, $|V| = n$, $|E| = m$

- Budujemy tablicę (macierz) M o rozmiarach $n \cdot n$.
- Wypełniamy tablicę M w sposób następujący:
 - Zerem - jeśli dwa wierzchołki nie są połączone krawędzią.
 - Jedynką - jeśli dwa wierzchołki są połączone.

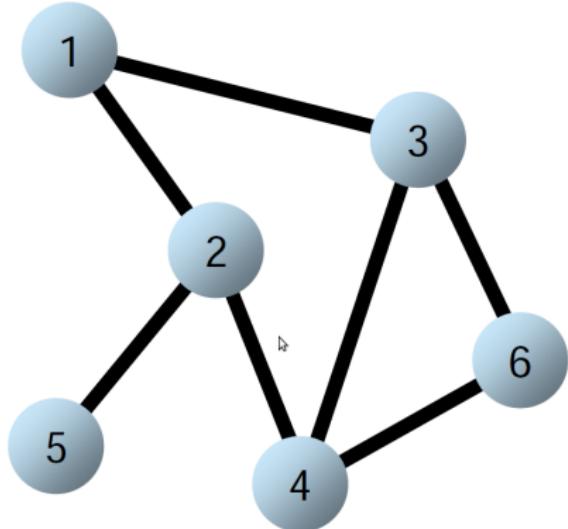
Uwagi: Dla grafów nieskierowanych macierz sąsiedztwa jest symetryczna, zatem implementując ją za pomocą tablic dynamicznych można ją zmniejszyć do połowy - zapisujemy tylko macierz dolno-(górnego)-trójkątną.

Złożoność pamięciowa: $O(n^2)$

Macierz sąsiedztwa - graf nieskierowany

	1	2	3	4	5	6
1		1	1			
2	1			1	1	
3	1			1		1
4		1	1			1
5		1				
6			1	1		

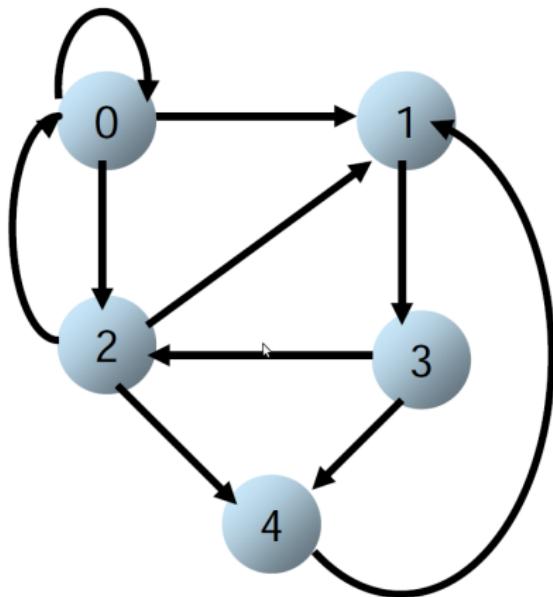
- Czas wstawienia: $O(1)$
- Czas usunięcia: $O(1)$
- Czas zapytania: $O(1)$



Macierz sąsiedztwa - graf skierowany

	0	1	2	3	4
0	1	1	1		
1				1	
2	1	1			1
3			1		1
4		1			

- Czas wstawienia: $O(1)$
- Czas usunięcia: $O(1)$
- Czas zapytania: $O(1)$



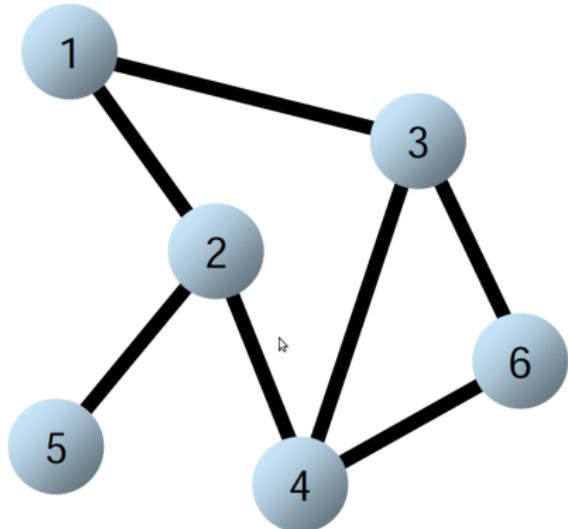
Dany jest graf $G = (V, E)$, $|V| = n$, $|E| = m$

- Budujemy listę dla każdego wierzchołka $v \in V$, w której przechowujemy zbiór wierzchołków połączonych krawędzią z v .
- Złożoność pamięciowa: $O(n + m)$.

Lista incydencji - graf nieskierowany

1:	2	3	
2:	1	4	5
3:	1	4	6
4:	2	3	6
5:	2		
6:	3	4	

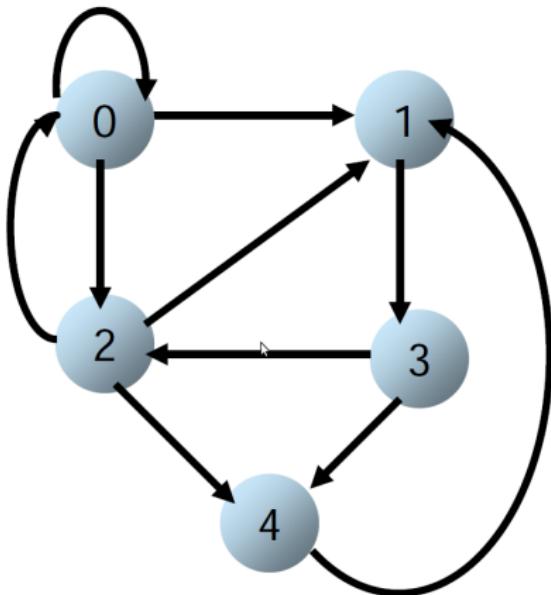
- Czas wstawienia: $O(1)$
- Czas usunięcia: czas wyszukiwania na liście uporzadkowanej
- Czas zapytania: czas wyszukiwania na liście uporzadkowanej



Lista incydencji - graf skierowany

0:	0	1	2
1:	3		
2:	0	1	4
3:	2	4	
4:	1		

- Czas wstawienia: $O(1)$
- Czas usunięcia: czas wyszukiwania na liście uporzadkowanej.
- Czas zapytania: czas wyszukiwania na liście uporzadkowanej.



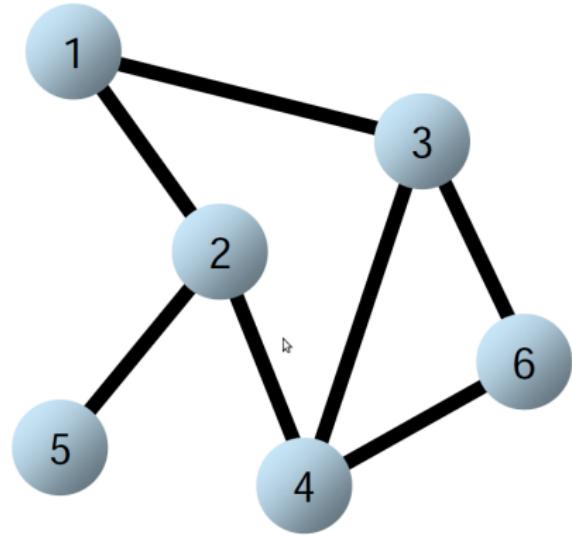
Lista Krawędzi

Dany jest graf $G = (V, E)$, $|V| = n$, $|E| = m$

- Lista krawędzi to lista, na której przechowujemy wszystkie krawędzie występujące w grafie.
- Zapisując przy pomocy tej reprezentacji graf, w którym występują krawędzie skierowane i nieskierowane należy w przypadku krawędzi nieskierowanej z u do v zapisać krawędź dwukrotnie: $u - v$ oraz $v - u$.
- Złożoność pamięciowa: $O(m)$.

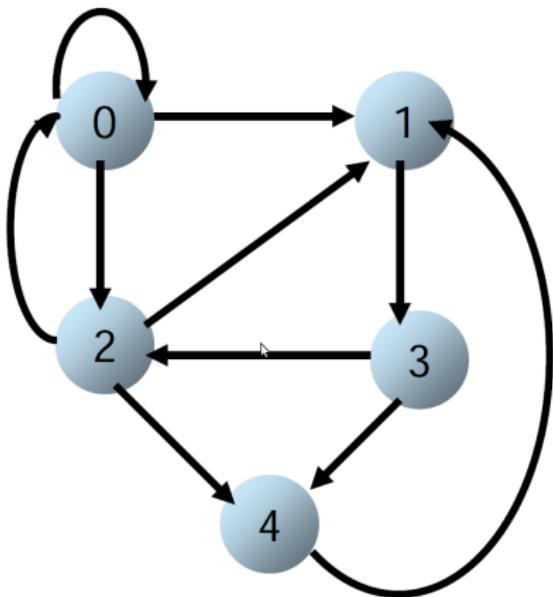
Lista krawędzi - graf nieskierowany

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



Lista krawędzi - graf skierowany

0 - 0
0 - 1
0 - 2
1 - 3
2 - 0
2 - 1
2 - 4
3 - 2
3 - 4
4 - 1



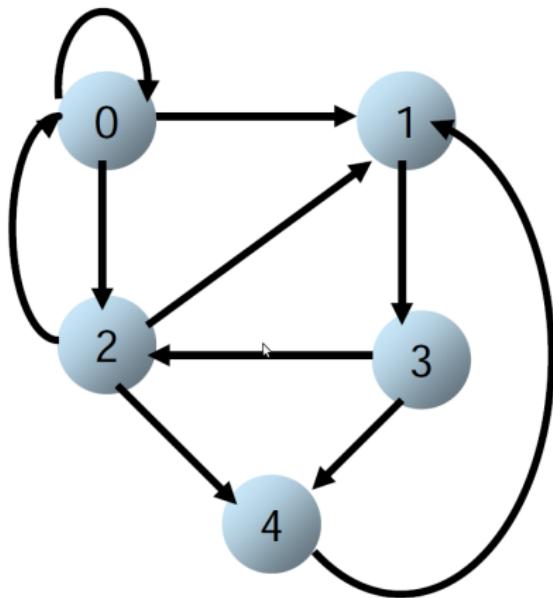
Macierz incydencji

Dany jest graf $G = (V, E)$, $|V| = n$, $|E| = m$

- Macierz incydencji to tablica o rozmiarach $n \cdot m$.
- Składa się ona z m kolumn i n wierszy:
 - jeśli krawędź wychodzi z danego wierzchołka, to piszemy w odpowiedniej kolumnie (-1),
 - jeśli krawędź wchodzi do danego wierzchołka, to piszemy (+1),
 - jeśli wierzchołek nie należy do krawędzi, to piszemy 0,
 - jeśli jest pętla własna, to piszemy 2.
- Złożoność pamięciowa $O(n \cdot m)$.

Macierz incydencji - graf skierowany

	0	1	2	3	4
0 - 0	2				
0 - 1	-1	1			
0 - 2	-1		1		
1 - 3		-1		1	
2 - 0	1		-1		
2 - 1		1	-1		
2 - 4			-1		1
3 - 2			1	-1	
3 - 4				-1	1
4 - 1		1			-1



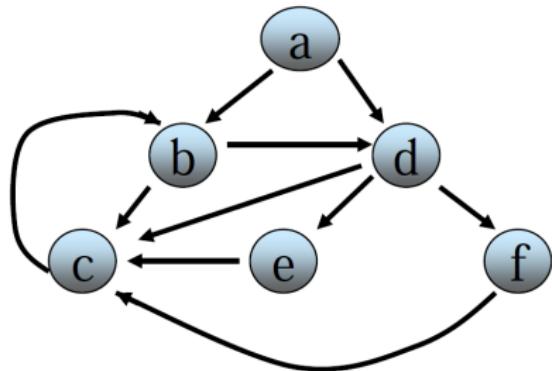
Macierz sąsiedztwa a lista sąsiedztwa

- Macierze sąsiedztwa są preferowanym sposobem reprezentacji grafów wówczas, gdy grafy są gęste, tzn. kiedy liczba krawędzi jest bliska maksymalnej możliwej ich liczby.
- Dla grafu skierowanego o n wierzchołkach maksymalna liczba krawędzi wynosi n^2 .
- Jeśli graf jest rzadki, to reprezentacja oparta na listach sąsiedztwa może pozwolić zaoszczędzić pamięć.

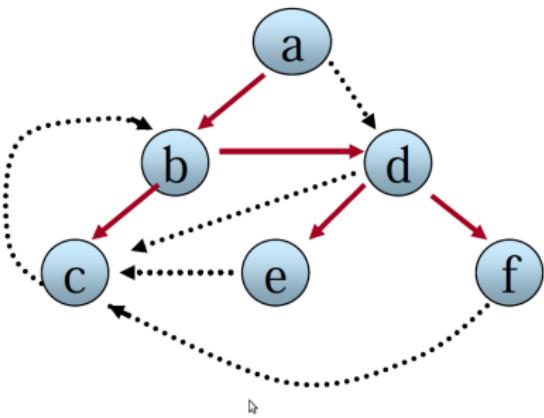
Algorytm przeszukiwania grafu w głąb - DFS

- Algorytm DFS to podstawowa metoda badania grafów skierowanych.
- Algorytm DFS wykorzystuje się do badania spójności grafu - jeśli procedura wywołana dla pierwszego wierzchołka "dotrze" do wszystkich wierzchołków grafu to graf jest spójny.
- Algorytm:
 - Wybrany wierzchołek umieść na stosie, zaznacz jako odwiedzony i przejdź do jego następnika. Następnik również umieść na stosie, zaznacz jako odwiedzony i przejdź do jego następnika.
 - Jeśli napotkany wierzchołek nie ma krawędzi incydentnych z nieodwiedzonymi wierzchołkami, usuń go ze stosu i pobierz ze stosu kolejny wierzchołek do przeszukania.
- W praktyce stosuje się zasadę, że jeśli przeszukiwany wierzchołek jest połączony krawędziami z wieloma wierzchołkami, wybiera się do przeszukania wierzchołek o najmniejszej liczbie porządkowej. Dlatego szukając kolejny nieodwiedzony następnik należy rozpoczynać od końca macierzy.

DFS - przykład



Jedno z możliwych drzew
przeszukiwania:

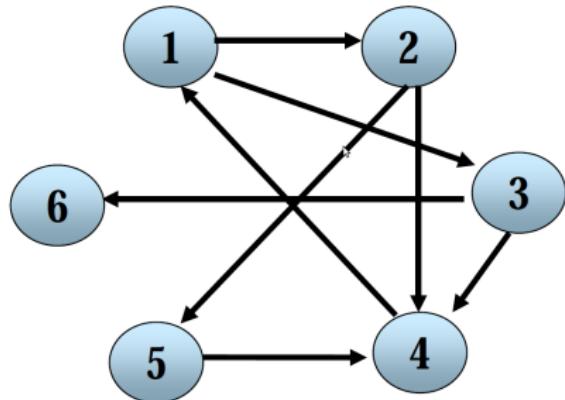


Przeszukiwanie grafu wszerz (BFS)

Aby przeszukać graf wszerz (BFS) należy zamiast stosu wykorzystać kolejkę do przechowywania wierzchołków, a kolejnych nieodwiedzonych następców szukać od początku macierzy.

BFS i DFS- przykład

- W wyniku wywołania procedury DFS dla grafu obok otrzymamy wierzchołki w następującej kolejności: 1,2,4,5,3,6.
- W wyniku wywołania procedury BFS dla grafu obok otrzymamy wierzchołki w kolejności: 1,2,3,4,5,6.



Graf - pewna implementacja

Definicja grafu

```
class Graph
{
    private:
        int n;          // liczba wierzchołków grafu
        bool** adj;    // matryca przylegania
    public:
        Graph(int); // inicjalizacja struktury Graph
        ~Graph();
        void addEdge(int a, int b); // dodanie krawedzi
        void display(); // wyświetlenie grafu
        void dfs(int); // wyszukiwanie "wglab"
        void bfs(int); // wyszukiwanie "wszerz"
        int getUnVisitedVertex(int, bool const []);
        friend void ::displayVertex(int);
};
```

Graf - pewna implementacja

Inicjalizacja struktury Graph

```
Graph::Graph( int n = 6 )
{
    this->n = n;
    this->adj = new (nothrow) bool* [n];
    for (int k = 0; k < n; ++k) {
        this->adj[k] = new (nothrow) bool [n];
    }
    for( int b = 0; b < n; ++b) {
        for( int a = 0; a < n; ++a) {
            this->adj[a][b] = false;
        }
    }
}
```

Graf - pewna implementacja

Dodanie krawędzi

```
void Graph::addEdge(int a, int b) {  
    if (a >= 0 && a < this->n &&  
        b >= 0 && b < this->n)  
    {  
        this->adj[a][b] = true;  
        this->adj[b][a] = true;  
    } else  
    {  
        cout << "Niepoprawne dane\n\n";  
    }  
}
```

Graf - pewna implementacja

Wyświetlenie grafu

```
void displayVertex(int a) {
    cout << (char)('A' + a);
}

void Graph::display() {
    cout << "\nKrawedzie grafu:\n";
    for (int row = 0; row < this->n; ++row) {
        for (int col = 0; col < this->n; ++col) {
            if (this->adj[row][col]) {
                cout << (char)('A' + row) << '-';
                cout << (char)('A' + col) << " ";
            }
        }
    }
    cout << endl;
}
```

Graf - pewna implementacja

nie odwiedzony wierzchołek

```
// zwraca nie odwiedzony wierzchołek przyległy do a
// zwraca -1, jeżeli takiego wierzchołka nie ma
int Graph::getUnVisitedVertex(int a, bool const visited [])
{
    for (int b = 0; b < this->n; ++b) {
        if (this->adj[a][b] && !visited[b]) {
            return b;
        }
    }
    return -1;
}
```

Graf - pewna implementacja

Wyszukiwanie "wgłęb"

```
void Graph::dfs(int a) // wyszukiwanie "wglab"
{
    bool* visited = new (nothrow) bool[this->n];
    for (int k = 0; k < this->n; ++k) visited[k] = false;
    STACK<int> s;
    visited[a] = true; // rozpoczęj od wierzchołka a
    displayVertex(a); // wyświetl wierzchołek
    s.push(a); // zapisz na stos
    while (!s.empty()) {
        // pobierz nie odwiedzony wierzchołek,
        // przyległy do szczytowego elementu stosu
        int b = this->getUnVisitedVertex(s.top(), visited);
        if (b == -1) { // jeżeli nie ma takiego wierzchołka,
            s.pop();
        } else {
            visited[b] = true; // oznacz wierzchołek jako odwiedzony
            displayVertex(b); // wyświetl wierzchołek
            s.push(b); // zapisz na stos
        }
    }
    delete[] visited;
}
```

Graf - pewna implementacja

wyszukiwanie wszerz

```
void Graph::bfs(int a) // wyszukiwanie "wszerz"
{
    bool* visited = new (nothrow) bool[this->n];
    for (int k = 0; k < this->n; ++k) visited[k] = false;
    Queue<int> q;
    visited[a] = true; // rozpocznij od wierzchołka a
    displayVertex(a); // wyświetl wierzchołek
    q.inject(a); // wstaw na koncu
    while (!q.empty()) { // do opróżnienia kolejki,
        int b = q.front(); // pobierz pierwszy wierzchołek
        q.eject(); // usun go z kolejki
        // dopóki ma nie odwiedzonych sąsiadów
        int c;
        while ((c = this->getUnVisitedVertex(b, visited)) != -1)
        {
            // pobierz sąsiada
            visited[c] = true; // oznacz
            displayVertex(c); // wyświetl
            q.inject(c); // wstaw do kolejki
        } // while
    } // while(kolejka nie jest pusta)
    delete [] visited;
}
```

1. When determining the efficiency of algorithm, the space factor is measured by

- a. Counting the maximum memory needed by the algorithm
- b. Counting the minimum memory needed by the algorithm
- c. Counting the average memory needed by the algorithm
- d. Counting the maximum disk space needed by the algorithm

2. The complexity of Bubble sort algorithm is

- a. $O(n)$
- b. $O(\log n)$
- c. $O(n^2)$
- d. $O(n \log n)$

3. Linked lists are best suited

- a. for relatively permanent collections of data
- b. for the size of the structure and the data in the structure are constantly changing
- c. for both of above situation
- d. for none of above situation

4. If the values of a variable in one module is indirectly changed by another module, this situation is called

- a. internal change
- b. inter-module change
- c. side effect
- d. side-module update

5. In linear search algorithm the Worst case occurs when

- a. The item is somewhere in the middle of the array
- b. The item is not in the array at all
- c. The item is the last element in the array
- d. The item is the last element in the array or is not there at all

6. For an algorithm the complexity of the average case is

- a. Much more complicated to analyze than that of worst case
- b. Much more simpler to analyze than that of worst case
- c. Sometimes more complicated and some other times simpler than that of worst case
- d. None or above

7. The complexity of merge sort algorithm is

- a. $O(n)$
- b. $O(\log n)$
- c. $O(n^2)$
- d. $O(n \log n)$

8. The complexity of linear search algorithm is

- a. $O(n)$
- b. $O(\log n)$
- c. $O(n^2)$
- d. $O(n \log n)$

9. When determining the efficiency of algorithm the time factor is measured by

- a. Counting microseconds
- b. Counting the number of key operations
- c. Counting the number of statements
- d. Counting the kilobytes of algorithm

10. Which of the following data structure is linear data structure?

- a. Trees
- b. Graphs
- c. Arrays
- d. None of above

11. The elements of an array are stored successively in memory cells because

- a. by this way computer can keep track only the address of the first element and the addresses of other elements can be calculated
- b. the architecture of computer memory does not allow arrays to store other than serially
- c. both of above
- d. none of above

12. Which of the following data structure is not linear data structure?

- a. Arrays
- b. Linked lists
- c. Both of above
- d. None of above

13. The Average case occur in linear search algorithm

- a. When Item is somewhere in the middle of the array
- b. When Item is not in the array at all
- c. When Item is the last element in the array
- d. When Item is the last element in the array or is not there at all

14. Two main measures for the efficiency of an algorithm are

- a. Processor and memory
- b. Complexity and capacity
- c. Time and space
- d. Data and space

15. Finding the location of the element with a given value is:

- a. Traversal
- b. Search
- c. Sort
- d. None of above

16. Which of the following case does not exist in complexity theory

- a. Best case
- b. Worst case
- c. Average case
- d. Null case

17. The operation of processing each element in the list is known as

- a. Sorting
- b. Merging
- c. Inserting
- d. Traversal

18. Arrays are best data structures

- a. for relatively permanent collections of data
- b. for the size of the structure and the data in the structure are constantly changing
- c. for both of above situation
- d. for none of above situation

19. Each array declaration need not give, implicitly or explicitly, the information about

- a. the name of array
- b. the data type of array
- c. the first data from the set to be stored
- d. the index set of the array

20. The complexity of Binary search algorithm is

- a. $O(n)$
- b. $O(\log_2 n)$
- c. $O(n^2)$
- d. $O(n \log n)$

On Understanding Social Propagation

Dariusz Krol¹²

¹Smart Technology Research Centre
Faculty of Science and Technology
Bournemouth University, United Kingdom



²Institute of Informatics
Wroclaw University of Technology, Poland



Wroclaw University of Technology

Chosun University, January 22, 2016

Outline

1 Introduction

2 Overview of propagation

3 Methodologies

4 Measurements

5 Conclusion

*„Never lose a holy curiosity”
Albert Einstein*

Outline

1 Introduction

2 Overview of propagation

3 Methodologies

4 Measurements

5 Conclusion

Behind propagation phenomenon

What is propagation phenomenon and why is it important?

- ① What: you hopefully know this... The term 'propagation' is used to describe methods for automatically disseminating action.
- ② Why: butterfly effect - a small action may contribute a significant and large-scale influence.

Where is propagation phenomenon applied?

- In complex and time-sensitive domains: transportation, real world networks, banking, industrial automation, robotics, online collaboration, multi-media development, and safety-critical applications.
- Common to distributed environments in the following forms: data migration for collaboration, adaptive task allocation and resource negotiation, mobile transfer and code update, message dispatching, routing algorithm, and self-organising computation.

Social propagation phenomenon

General idea

The general idea behind *social propagation* is that individuals interact repeatedly under the reciprocal influence of other individuals, modelled as a *social influence score*, which may often generate a flood of action across the connections. Social propagation occurs in various forms.

Elementary social examples include:

- **structure mining:** community detection, locating and repairing faults, finding effectors and maximizing influence,
- **process activity:** joining events or groups, rumor dissemination, forwarding memes, hashtags re-tweeting, and purchasing products.

Francesco Bonchi. Influence propagation in social networks: A data mining perspective. IEEE Intelligent Informatics Bulletin, 12(1):8–16, 2011

Benjamin Doerr, Mahmoud Fouz, and Tobias Friedrich. Why rumors spread so quickly in social networks. Commun. ACM, 55(6):70–75, 2012

Propagation phenomenon in other domains

More examples

- From epidemiology the propagation is mimicked the spread of a contagious disease in fully decentralized manner.
- In computer science, the propagation method involves transfer, reproduction or multiplication of data resulting in a repeatable action in the context of pursuing a defined goal.
- In software engineering, error propagation may occur between system components and between functions in a single program.
- Every scientific measurement is subjected to errors derived from various sources: problem of definition, scale precision, intrinsic random uncertainty and systematic wrong values.
- For estimating the reliability of a complex system, error propagation is the problem of analysing uncertainty in estimating the distribution of the outcomes given (known) distribution of all the input parameters.

Error propagation phenomenon

General idea

An error is a difference between reality and the representation of reality. It includes deviations from accuracy, mistakes or faults, statistical variations, fallacy in reasoning, anomaly in software, and social misbehaviour. Errors are defined in different ways:

- Any demand or result that has occurred in a system and does not meet system specification.
- Any state reflecting a fault during the production process.
- Any difference between the current state and the expected state.

Several similar terms are used by researchers to describe the 'error' concept, including failure, fault, defect, flaw, degradation, exception, bug, glitch, conflict and uncertainty,

Error propagation phenomenon

Types of errors in distributed system

- The omission error manifests itself by not sending a message which should have been sent according to protocol. It can be a result of connection error.
- The commission error is manifested by the invalid communication behaviour consisting in sending a message not meant to be sent. It can be a result of a deliberate attack.
- The contamination error manifests itself by invalid data sent. The source of this symptom might be a fault application sending a contaminated data.
- The crash error is caused by the failure of the network component and consists in stopping its communication activity.

The error propagation is the series of connected error events, e.g. spreading of faulty behaviour.

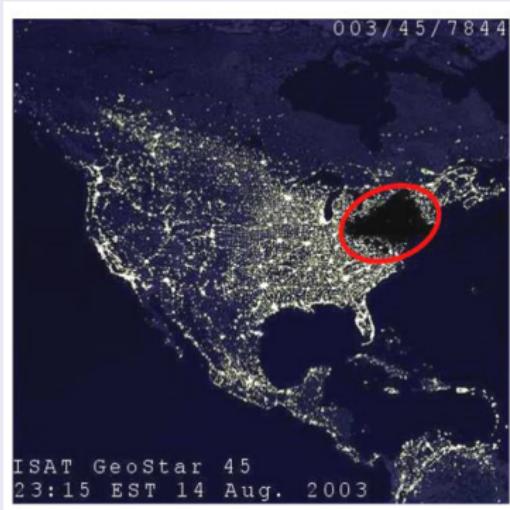
Error propagation phenomenon

More issues

- To find the optimal recovery points of the processes suffering from error propagation (Fault Tolerance Analysis).
- To generate the exception propagation paths, which includes the origin of exceptions.
- To model an intrusion and its propagation path through various attack schemas.
- To analyse the propagation of roundoff errors throughout a calculation.
- To design robust algorithms.

Examples

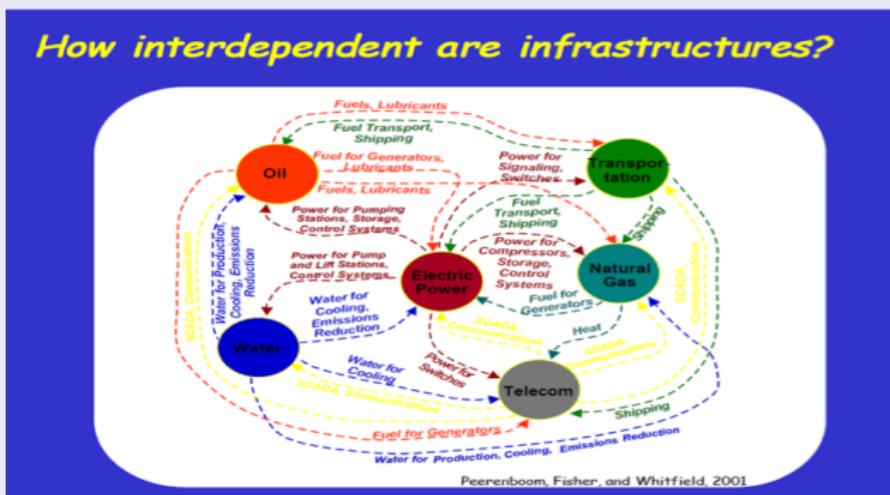
Blackouts



Examples

Cascading failures

Failure of a part of a system can trigger the failure of successive parts.



Problems

Frequently raised questions

- What properties of the propagation process are essential for efficiency and robustness?
- How does the propagation process normally proceed? How to best fit a propagation model and efficiency indicators?
- Is this true that the topology has the influence on the propagation phenomenon? What is fundamental to define the exact influence of topology on error propagation's course?
- Is this true, that the higher connectivity of the system's communication graph, the higher error tolerance of the system?
- How quickly, widely and robustly does social propagation spread?
- How to design the most error resistant architectures in complex environments? Which topology is the better of the other?

Opportunities

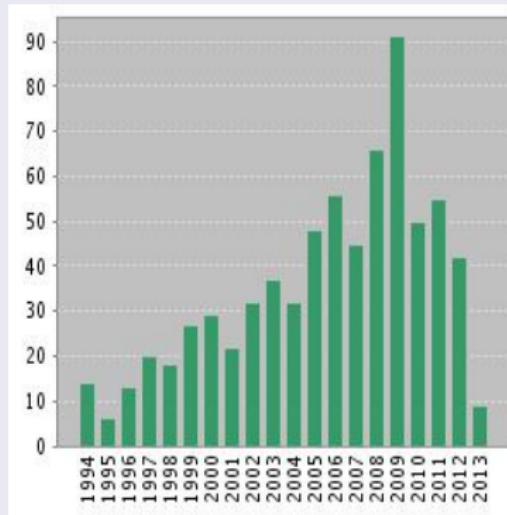
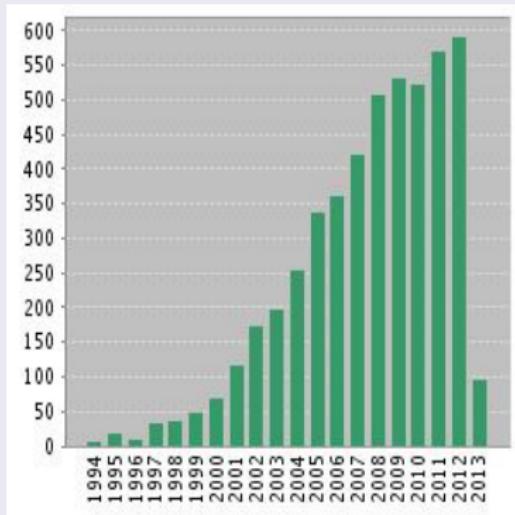
Challenges

- Increased complexity of networks
- Coupling between networks
- Interdependencies in given networks
- Dynamic nature and spatial properties of the networks
- Interrelationships between structure, dynamics and function of networks
- Lack of centralized control
- Need for survivability or rapid perish
- Potential errors and attacks - designing 'better' networks, not only robust to the random loss of nodes, but also less vulnerable to the selective loss of the most important ones

Popularity of published research on propagation

Web of Science®

Topic=(propagation phenomenon) Areas=(ALL|COMPUTER SCIENCE)



Outline

1 Introduction

2 Overview of propagation

3 Methodologies

4 Measurements

5 Conclusion

Features of a propagation process

Principles

Topology-dependent. Propagation utilizes to the utmost the underlying structure of a network. Most real networks are characterized with the *assortative mixing*, *broad degree distributions*, over time *shrinking diameters*, and *aging effect*. These features can significantly speed up the process.

Complexity. Real data are often interconnected, have overlapping communities and are coupled across time through processes. When nodes in one network depend on nodes in another, a small disturbance in one network can cascade through the entire system often growing to sufficient size to collapse it.

Features of a propagation process

Principles

Fast influence by ties. Propagation extends Granovetter's hypothesis that the majority of influence is collectively generated by weak ties, although strong ties are individually more influential. Very often **action spreads only partially overlapping the network, but extremely fast.**

Push-pull activity. The standard protocol is based on symmetric push-pull activity. It pushes the information in case it has, and pulls the information in case the neighbor has. A common form of positive and negative propagation is **spreading of breaking news, rumors, fads, beliefs, sentiments and norms.**

Features of a propagation process

Principles

Survive or perish. Individuals tend to adopt the behavior of their peers, so first propagation happens locally in their neighborhood. Then, this behaviour might become global and survive or decay and finally perish as it crosses the network.

Epidemic-type and cascading-type. Propagation can be conceptualized either as an *epidemic-type dynamic*, where a node in an infected state may infect a neighbor independently of the status of the other nodes, or as a *cascading-type dynamic*, where the change may depend on the reinforcement caused by changes in other nodes.

Our previous research

Distributed sum algorithm

Every entity e_i is given an initial random value $k(e_i)$ known only to this entity. After finite execution time each of the entities should know the exact sum of all initial values $S = \sum_{e \in \mathcal{E}} k(e_i)$. In the first step the entity e_i broadcasts its input value $k(e_i)$ to all of its neighbors. Then it waits for messages from the other entities. Incoming message is processed only if previously there were no other messages coming from this sender. Message processing consists in increasing internal sum $o(e_i)$ by the value received and broadcasting the received value. The process terminates when the entity does not change its internal sum, after all entities have fixed their states.

Our previous research

Errors

The Crash error can occur before the faulty entity receives a message. If this happens, the entity immediately finishes all of its activities. The Omission error happens during the message broadcasting phase. If it occurs, the entity does not send the message to one of its neighbors. The Commission error takes place after all of the messages have been sent. If it occurs, the faulty entity creates a random message and sends it to one of its neighbors. The Contamination error causes the faulty entity to randomly modify one message sent to one of its neighbors during the broadcasting step.

Our previous research

Measures

The damage assessment was done by using the fraction of the entities' number affected by the error (having the internal state at the end of execution different than the \mathcal{S} value). The *Error Percentage* value can be defined as $E_p = \frac{|F|}{|\mathcal{E}|}$, where

$F = \{e \in \mathcal{E} : o(e) \neq \mathcal{S}\}$ is the set of error-affected entities. Also the algorithm operation time was used as a measure, because some errors did not directly affect the system's state but they did lengthened the execution time.

Our previous research

Quantitative results

Topology	Error Type					Avg.
	None	Omi.	Com.	Con.	Crash	
Full	0	0	0	0.04	0.51	0.11
Random	0	0.05	0.16	0.28	0.55	0.21
Ring	0	0	0.39	0.44	0.51	0.27
Star	0	0.49	0.31	0.46	0.58	0.37
Tree	0	0.52	0.31	0.52	0.69	0.41
Avg.	0	0.21	0.23	0.35	0.57	0.27

Average error percentage relationship to topology and error type.

Our previous research

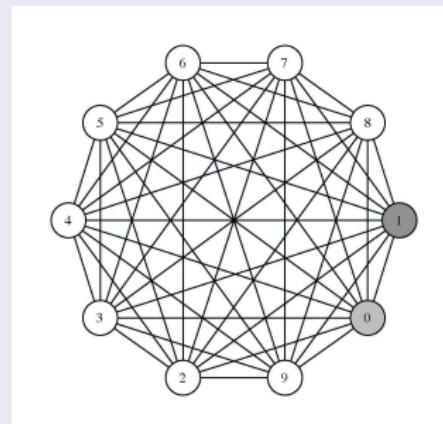
Quantitative results

- Contamination in regular structure
- Commission in random structure
- Contamination in social structure
- Commission in regular structure
- Commission in hierarchical structure

Our previous research

Quantitative results

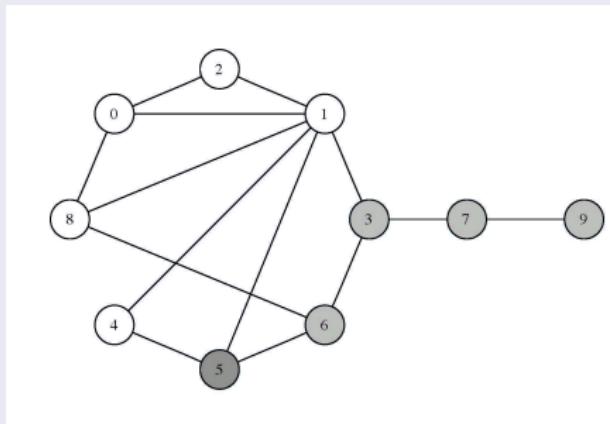
- Contamination in regular structure
- Commission in random structure
- Contamination in social structure
- Commission in regular structure
- Commission in hierarchical structure



Our previous research

Quantitative results

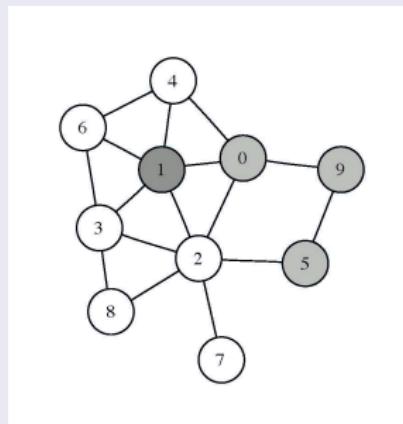
- Contamination in regular structure
- Commission in random structure
- Contamination in social structure
- Commission in regular structure
- Commission in hierarchical structure



Our previous research

Quantitative results

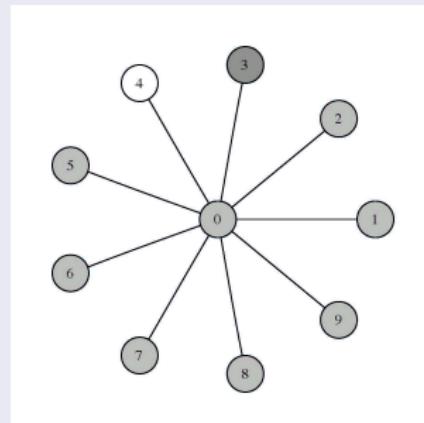
- Contamination in regular structure
- Commission in random structure
- **Contamination in social structure**
- Commission in regular structure
- Commission in hierarchical structure



Our previous research

Quantitative results

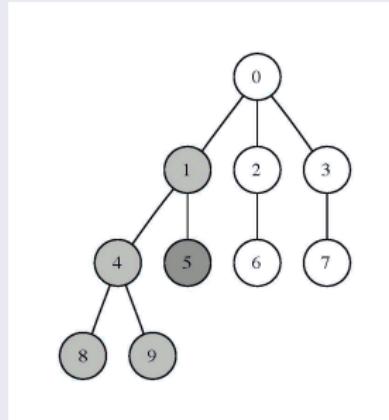
- Contamination in regular structure
- Commission in random structure
- Contamination in social structure
- Commission in regular structure
- Commission in hierarchical structure



Our previous research

Quantitative results

- Contamination in regular structure
- Commission in random structure
- Contamination in social structure
- Commission in regular structure
- **Commission in hierarchical structure**



Our previous research

Conclusions

- The topologies can be ordered from the most to the least immune to error propagation in the undermentioned way: (1) full topology, (2) random topology, (3) ring topology, (4) star topology and (5) tree topology.
- The error types can be ordered form the least to the most harmful: (1) omission, (2) commission, (3) contamination, (4) crash.

Outline

1 Introduction

2 Overview of propagation

3 Methodologies

4 Measurements

5 Conclusion

Propagation strategy

Propagation-based techniques play an important role in maintaining existing networks, e.g. synchronization in electric power grids, prediction of complex system behaviour, resource discovery and monitoring, locating biological invasions and assessing damage, virus propagation control and containment, decomposition and immunization of social and large scale infrastructure networks.

Strategy

Propagation strategy based on biology-inspired systems and mechanisms

- Living cells provide outstanding examples of collective behavior underpinned by networks of interactions. Molecular-level networks coordinate countless biological functions and allow the cells to respond to ever-changing environments. A newly encountered signal, such as excess sodium, may propagate through the network and alter cellular behavior. But even in the absence of external signals, interactions among genes and proteins can lead to multiple steady-state and dynamic equilibria.

Propagation strategy based on epidemic spreading

- Epidemic spreading is frequently used as an analogy to understand the information dissemination in wireless ad hoc networks. Information dissemination in this context can refer to the distribution of information particles (as usually provided by ad hoc routing techniques) or to the spread of viruses in the Internet or on mobile devices.

How does the propagation process normally proceed?

- How to aptly adopt the propagation mechanism to make the information (action) spreading faster and wider?
- How to stop diffusing the undesirable things which can threaten individuals or organizations?

Input: graph G , threshold θ_i for node, seed nodes A_0 , time limit T

Output: *triplets*

```
active, triplets  $\leftarrow \emptyset$ ;  
waiting  $\leftarrow A_0$ ;  
 $t \leftarrow 0$ ;  
while termination condition not satisfied do  
    foreach node  $i \in \text{waiting}$  do  
         $t \leftarrow t + 1$ ,  $\text{waiting} \leftarrow \text{waiting} \setminus \{i\}$ ,  $\text{active} \leftarrow \text{active} \cup \{i\}$ ;  
        propagation loop with threshold condition  
    end  
end
```

Algorithm 1: Generic propagation algorithm

How does the propagation process normally proceed?

In the algorithm, we use propagation loop with threshold condition, that determines how to select next node to activate it in the process to maximize the spread. In the push variant of propagation loop and condition (Algorithm 2), propagator one-to-many using \deg^+ operator is required, e.g., sending photo to all my group members.

```
foreach node  $j \in \deg^+(i)$  do
    if  $\alpha(i, j) \geq \theta_j \wedge j \notin active, waiting$  then
         $waiting \leftarrow waiting \cup \{j\};$ 
         $triplets \leftarrow triplets \cup \{i, j, t\};$ 
    end
end
```

Algorithm 2: Push variant of propagation loop

How does the propagation process normally proceed?

In the pull variant, see Algorithm 3, each node's tendency to become active increases monotonically as more of its neighbors become active. This time the propagator requires a many-to-one operator using deg^- which corresponds to a decision taken by an expert committee.

```
foreach node  $j \in \text{deg}^-(i)$  do
    if  $\sum_{k \in \text{deg}^-(j) \wedge k \in \text{active}} \alpha(k, j) \geq \theta_j \wedge j \notin \text{active}, \text{waiting}$  then
        |  $\text{waiting} \leftarrow \text{waiting} \cup \{j\};$ 
        |  $\text{triplets} \leftarrow \text{triplets} \cup \{i, j, t\};$ 
    end
end
```

Algorithm 3: Pull variant of propagation loop

Strategy

How does the propagation process normally proceed?

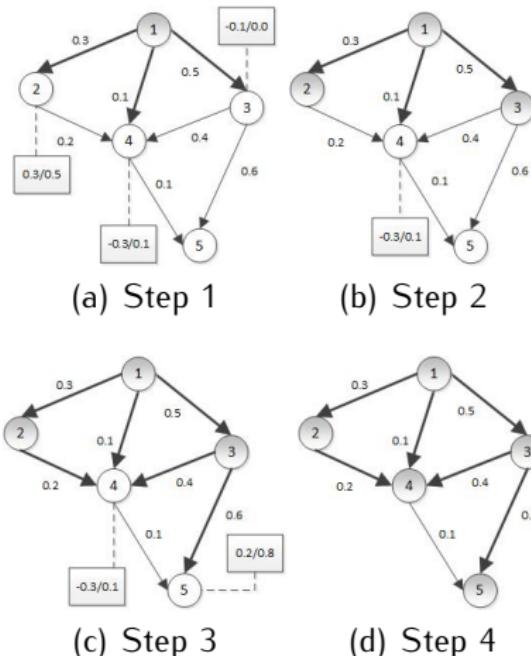


Figure: Preference-pull variant of propagation algorithm

Outline

1 Introduction

2 Overview of propagation

3 Methodologies

4 Measurements

5 Conclusion

How do we find out what we should measure?

- Know the properties of the network you are studying - network analysis
- Generate many of them using appropriate stochastic models - network generation
- Record several measures, and the value of the outcome process - optimization
- Analyze which measures are linked to the outcome - data mining

Propagation phenomenon

- Size, complexity and diversity of the network makes it very difficult to understand cause-effect relationships
- Measurement is necessary for understanding current system behavior and how new systems will behave. How, when, where, what do we measure? Before any measurements can take place one must determine what to measure
- There are many commonly used network performance characteristics Latency, Throughput, Response time, Arrival rate, Utilization, Bandwidth, Loss Routing, Reliability
- Measurement is meaningless without careful analysis, reproducibility is often essential
- Many tools have been developed to measure/monitor general characteristics of network performance

Propagation measures for complex networks

Measure	Meaning	Definition
σ_1 : propagation distance $d_{ij}^{p(G)}(t_1, t_2)$	The number of steps it takes for action to spread from node i to node j .	
σ_2 : propagation centrality $C_k^{p(G)}(t_1, t_2)$	The number of paths a propagation process can pass through a given node.	$\sum_i \sum_j \frac{\ (i, k, j) \ }{\ (i, j) \ }, i \neq j$
ϕ_1 : propagation efficiency $E^{p(G)}(t_1, t_2)$	The efficiency of propagation process from one node to all others.	$\frac{1}{N^{p(G)}(N^{p(G)} - 1)} \sum_{i \neq j \in 1..N} \frac{1}{d_{ij}^{p(G)}(t_1, t_2)}$
ϕ_2 : propagation robustness $R_{t_d}^{p(G)}(t_1, t_2)$	The ability of propagation process to maintain its activity after damage at time step t_d .	$1 - \frac{ E^{p(G)}(t_1, t_d) - E^{p(G)}(t_d, t_2) }{E^{p(G)}(t_1, t_2)}$

Outline

1 Introduction

2 Overview of propagation

3 Methodologies

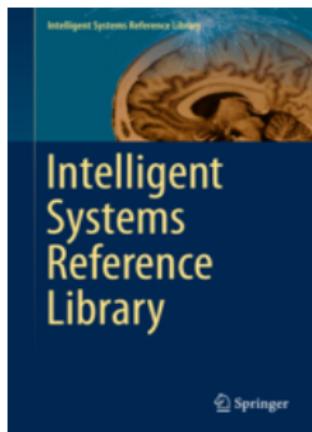
4 Measurements

5 Conclusion

Our future work

- ① Multiple important questions are still open, like understanding the tipping point of epidemics, predicting who wins among competing services/viruses/products, developing effective algorithms for immunization, and building more realistic propagations models while analyzing numerous real data sets.
- ② One interesting but to date unsolved problem is how to learn on the fly the time-varying elements of propagation by mining the present and archived log of past propagations.
- ③ Another open issue is the low efficiency of greedy algorithms to compute the influence problem to large networks. Improving the greedy algorithm is difficult, so this leads to a second possibility - the quest for appropriate heuristic.
- ④ Remains more: maximizing robustness via new structures...

Book already published



"Propagation Phenomena in Real World Networks"

A book edited by
Dariusz Krol, Damien Fay, Bogdan Gabrys
already published by Springer within the
Intelligent Systems Reference Library series.

*** ***

Topics in this publication include the following:

- information filtering and diffusion,
- activity coordination, collaboration and cooperation,
- knowledge propagation and integrity maintenance,
- epidemic spreading including worm propagation,
- cascading failures analysing and the damage assessment,
- big data migration and metadata evolution,
- identifying influential spreaders,
- expertise location and team formation,
- collaborative decision-making techniques,
- collective behaviour of colonies,
- anomaly detection, prevention and immunization,
- robustness/reliability estimation and improvement, and
- predictive models and techniques.

Acknowledgments

This work was supported by the European Commission
under grant PIEF-GA-2010-274375

Thank you for your attention

Grafy B

na podstawie wykładu prof. dr hab. Elżbieta Richter-Wąs z UJ

Graf to jest relacja binarna.

Dla grafów mamy ogromne możliwości wizualizacji jako zbiór punktów (zwanych wierzchołkami) połączonych liniami lub strzałkami (nazwanych krawędziami). Pod tym względem graf stanowi uogólnienie drzewiastego modelu danych. Podobnie jak drzewa, grafy występują w różnych postaciach: grafów skierowanych i nieskierowanych lub etykietowanych i nieetykietowanych.

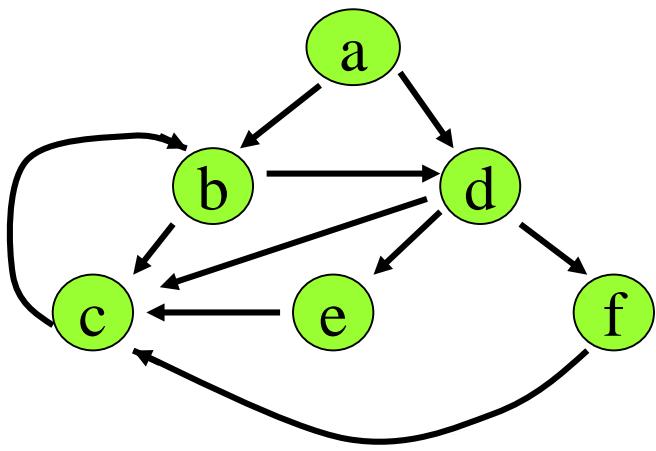
Grafy są przydatne do analizy szerokiego zakresu problemów: obliczenie odległości, znajdowanie cykliczności w relacjach, reprezentacji struktury programów, reprezentacji relacji binarnych, reprezentacji automatów i układów elektronicznych.

Algorytm przeszukiwania w głąb

Jest to podstawowa metoda badania grafów skierowanych. Bardzo podobna do stosowanych dla drzew, w których startuje się od korzenia i rekurencyjnie bada wierzchołki potomne każdego odwiedzonego wierzchołka. Trudność polega na tym że w grafie mogą pojawiać się cykle... należy wobec tego znaczyć wierzchołki już odwiedzone i nie powracać więcej do takich wierzchołków.

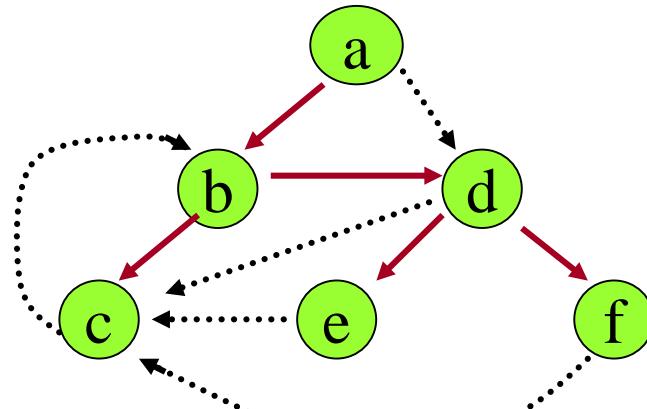
Z uwagi na fakt, że w celu uniknięcia dwukrotnego odwiedzenia tego samego wierzchołka jest on odpowiednio oznaczany, graf w trakcie jego badania zachowuje się podobnie do drzewa. W rzeczywistości można narysować drzewo, którego krawędzie rodzic-potomek będą niektórymi krawędziami przeszukiwanego grafu G .

Takie drzewo nosi nazwę drzewa przeszukiwania w głąb (ang. depth-first-search-first) dla danego grafu.

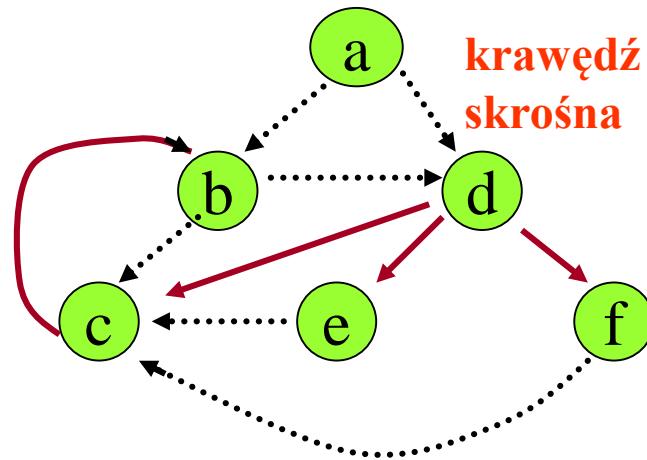


Graf skierowany

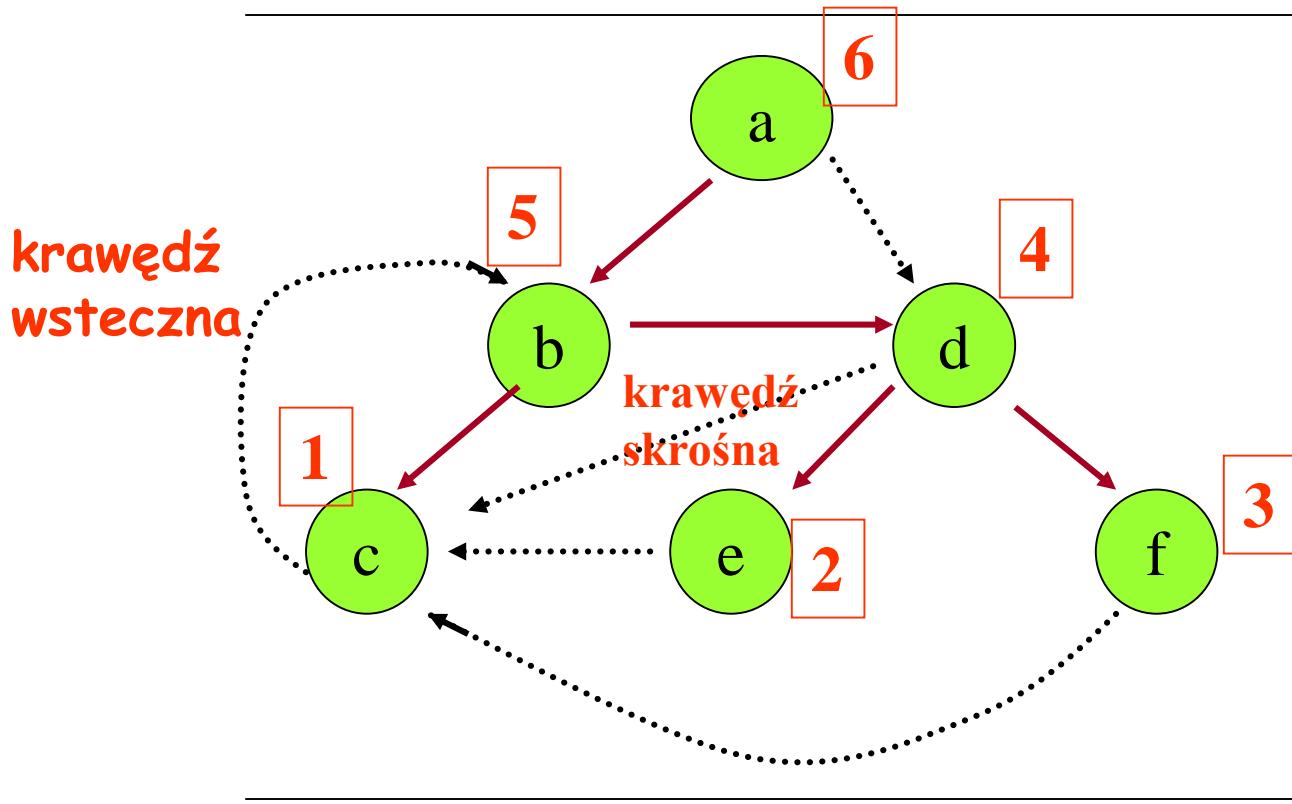
Las przeszukiwania:
dwa drzewa o korzeniach a, d



Jedno z możliwych drzew przeszukiwania



Las przeszukiwania w głąb.



Po (podczas) konstruowaniu drzewa przeszukiwania w głąb można ponumerować jego wierzchołki w kolejności wstecznej (ang. postorder).

Rekurencyjna funkcja przeszukiwania w głąb: void dfs

```
enum MARKTYPE {VISITED, UNVISITED};  
typedef struct{  
    enum MARKTYPE mark;  
    LIST successors;  
} GRAPH[MAX];
```

```
typedef struct CELL *LIST;  
struct CELL {  
    NODE nodeName;  
    LIST next;  
};
```

```
void dfs(NODE u, GRAPH G)  
{  
    LIST p; /* lista sąsiedztwa dla wierzchołka u */  
    NODE v; /* wierzchołek w komórce wskazywanej przez p */  
  
    G[u].mark = VISITED;  
    p = G[u].successors;  
    while (p != NULL) {  
        v = p->nodeName;  
        if (G[v].mark == UNVISITED) dfs(v, G);  
        p = p->next;  
    }  
}
```

Znajdowanie cykli w grafie skierowanym

Podczas przeszukiwania w głąb grafu skierowanego G można wszystkim wierzchołkom przypisać numery zgodne z kolejnością wsteczną w czasie rzędu $O(m)$.

Krawędzie wsteczne to takie dla których początki są równe lub mniejsze końcom ze względu na numerację wsteczną.

Zawsze gdy istnieje krawędź wsteczna w grafie musi istnieć cykl.
Prawdziwe jest również twierdzenie odwrotne.

Aby stwierdzić czy w grafie występuje cykl należy przeprowadzić numerację wsteczną a następnie sprawdzić wszystkie krawędzie.

Całkowity czas wykonania testu cykliczności to $O(m)$, gdzie m to większa z wartości liczby wierzchołków i liczby krawędzi.

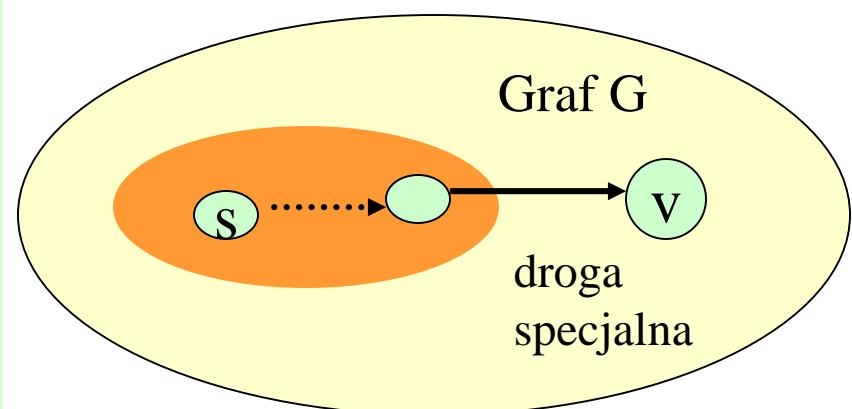
Algorytm Dijkstry znajdowania najkrótszych dróg.

Rozpatrujemy graf G (skierowany lub nieskierowany), w którym wszystkie krawędzie zaetykietowano wartościami reprezentującymi ich długości.

Długość (ang. distance) danej drogi stanowi wartość sumy etykiet związanych z nią krawędzi. Minimalna odległość z wierzchołka u do wierzchołka v to minimalna długość którejś z dróg od u do v .

Traktujemy wierzchołek s jako wierzchołek źródłowy. W etapie pośrednim wykonywania algorytmu w grafie G istnieją tzw. wierzchołki ustalone (ang. settled), tzn. takie dla których znane są odległości minimalne. W szczególności zbiór takich wierzchołków zawiera również wierzchołek s .

Dla nieustalonego wierzchołka v należy zapamiętać długość najkrótszej drogi specjalnej (ang. soecial path) czyli takiej która rozpoczyna się w wierzchołku źródłowym, wiedzie przez ustalone wierzchołki, i na ostatnim etapie przechodzi z obszaru ustalonego do wierzchołka v .



Dla każdego wierzchołka u zapamiętujemy wartość $\text{dist}(u)$.

Jeśli u jest wierzchołkiem ustalonym, to $\text{dist}(u)$ jest długością najkrótszej drogi ze źródła do wierzchołka u . Jeśli u nie jest wierzchołkiem ustalonym, to $\text{dist}(u)$ jest długością drogi specjalnej ze źródła do u .

Na czym polega ustalanie wierzchołków:

(1) znajdujemy wierzchołek v który jest nieustalony ale posiada najmniejszą $\text{dist}(v)$ ze wszystkich wierzchołków nieustalonych

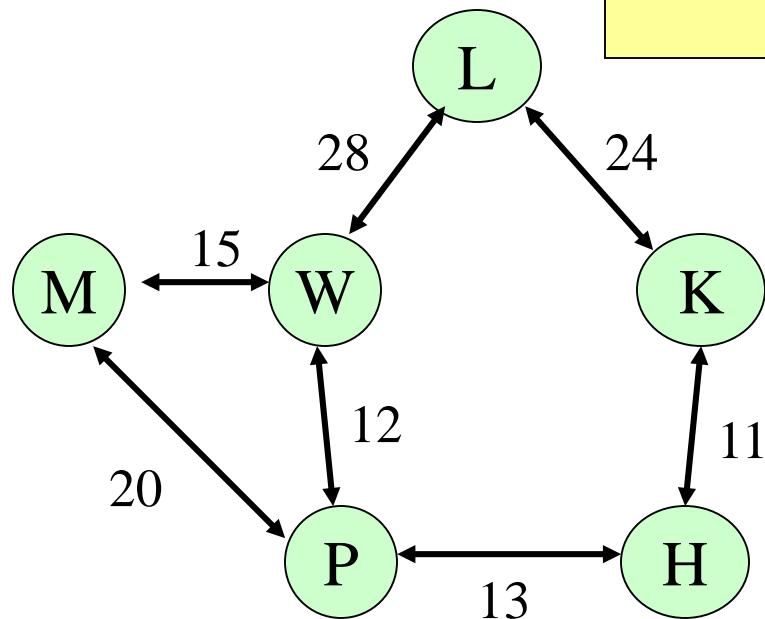
(2) przyjmujemy wartość $\text{dist}(v)$ za minimalną odległość z s do v

(3) dostosowujemy wartości wszystkich $\text{dist}(u)$ dla innych wierzchołków które nie są ustalone, wykorzystując fakt, że wierzchołek v jest już ustalony. Czyli porównujemy stare $\text{dist}(u)$ z wartością $\text{dist}(v) + \text{etykieta}(v \rightarrow u)$ jeżeli taka krawędź istnieje.

Czas wykonania algorytmu jest $O(m \log n)$.

Etapy wykonania algorytmu Dijkstry

MIASTO	ETAPY ustalania wierzchołków				
	(1)	(2)	(3)	(4)	(5)
H	0*	0*	0*	0*	0*
P	13	13	13*	13*	13*
M	INF	INF	33	33	33*
W	INF	INF	25	25*	25*
L	INF	35	35	35	35
K	11	11*	11*	11*	11*



Indukcyjny dowód poprawności algorytmu

W celu wykazania poprawności algorytmu Dijkstry należy przyjąć, że etykiety krawędzi są nieujemne. Indukcyjny dowód poprawności względem k prowadzi do stwierdzenia że:

- (a) dla każdego wierzchołka ustalonego u, wartość $\text{dist}(u)$ jest minimalną odlegością z s do u, a najkrótsza droga do u składa się tylko z wierzchołków ustalonych.
- (b) dla każdego nieustalonego wierzchołka u, wartość $\text{dist}(u)$ jest minimalną długością drogi specjalnej z s do u (jeśli droga nie istnieje wartość wynosi INF).

Podstawa:

Dla $k=1$ wierzchołek s jest jedynym wierzchołkiem ustalonym. Inicjalizujemy $\text{dist}(s)$ wartością 0, co spełnia warunek (a).

Dla każdego innego wierzchołka u, $\text{dist}(u)$ jest inicjalizowane wartością etykiety krawędzi $s \rightarrow u$, o ile taka istnieje. Jeżeli nie istnieje, wartością inicjalizacji jest INF. Zatem spełniony jest również warunek (b).

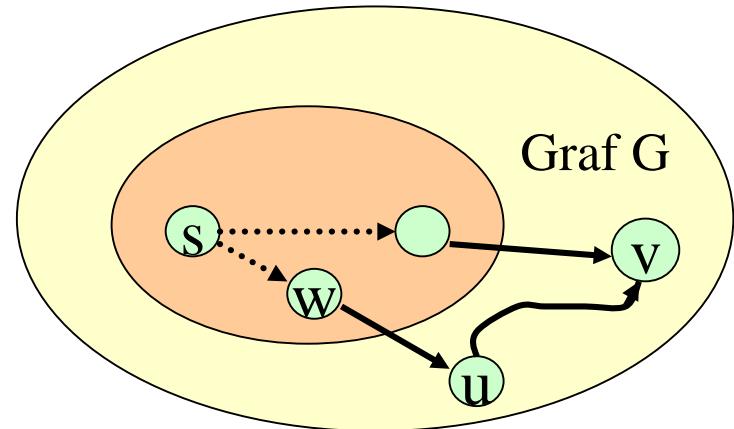
Indukcja:

Załóżmy, że warunki (a) i (b) za spełnione po ustaleniu k wierzchołków oraz niech v będzie (k+1) ustalonym wierzchołkiem. Warunek (a) jest wciąż spełniony ponieważ $\text{dist}(v)$ jest najmniejszą długością drogi z s do v.

Załóżmy, że tak nie jest. Musiała by więc istnieć hipotetyczna krótsza droga do v wiodąca przez w i u. Jednakże wierzchołek v został obrany jako k+1 ustalony, co oznacza, że w tym momencie $\text{dist}(u)$ nie może być mniejsze od $\text{dist}(v)$, gdyż wówczas jako (k+1) wierzchołek wybrany zostałby wierzchołek u.

Na podstawie warunku (b) hipotezy indukcyjnej wiadomo, że $\text{dist}(u)$ jest minimalna długością drogi specjalnej wiodącej do u. Jednak droga z s przez w do u jest drogą specjalną, tak więc jej długość równa jest co najmniej $\text{dist}(u)$. Stąd domniemana krótsza droga z s do v wiodąca przez w i u ma długość równą co najmniej $\text{dist}(v)$, ponieważ pierwsza jej część, - z s do u - ma długość $\text{dist}(u)$, a $\text{dist}(u) \geq \text{dist}(v)$. Stąd warunek (a) jest spełniony dla k+1 wierzchołków.

Hipotetyczna krótsza droga do v wiodąca przez w i u.



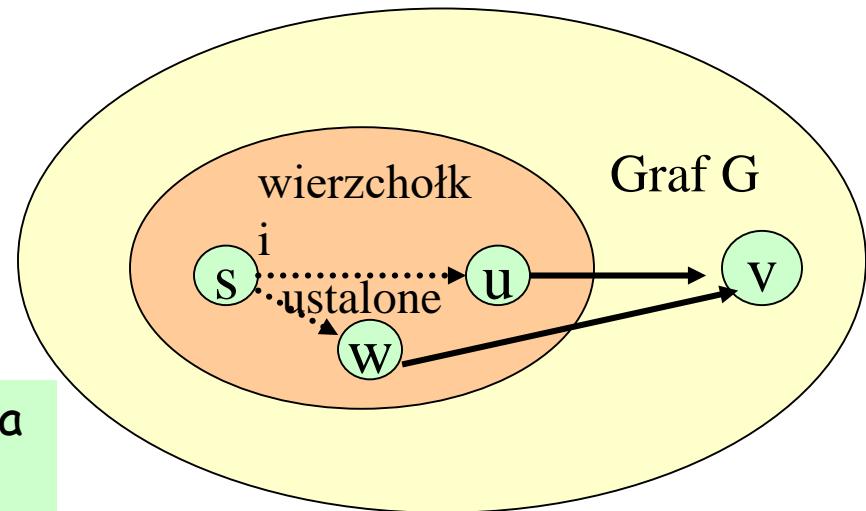
Teraz należy pokazać, że warunek (b) jest spełniony po dodaniu do wierzchołków ustalonych wierzchołka v . Weźmy pod uwagę pewien wierzchołek u , który wciąż pozostaje nieustalony po dodaniu v do wierzchołków ustalonych. W najkrótszej drodze specjalnej do u musi istnieć pewien wierzchołek przedostatni; wierzchołkiem tym może być zarówno v , jak i pewien inny wierzchołek w .

Przyjmijmy, że wierzchołkiem przedostatnim jest v . Długość drogi z s przez v do u wynosi $\text{dist}(v) + \text{wartość etykiety } v \rightarrow u$.

Przyjmijmy, że wierzchołkiem przedostatnim jest w . Na podstawie warunku (a) hipotezy indukcyjnej można stwierdzić, że najkrótsza droga z s do w składa się jedynie z wierzchołków, które zostały ustalone przed v , stąd wierzchołek v nie występuje w tej drodze. A więc długość drogi specjalnej do u się nie zmienia po dodaniu v do wierzchołków ustalonych.

Ponieważ w momencie ustalania wierzchołka v przeprowadzona jest operacja dostosowywania $\text{dist}(u)$, warunek (b) jest spełniony.

Dwie możliwości określenia przedostatniego wierzchołka w drodze specjalnej do u .



Jeśli potrzebne jest poznanie minimalnych odległości między wszystkimi parami wierzchołków w grafie o n wierzchołkach, które posiadają etykiety o wartościach nieujemnych, można uruchomić algorytm Dijkstry dla każdego z n wierzchołków jako wierzchołka źródłowego. Czas wykonania algorytmu Dijkstry wynosi $O(m \ln n)$, gdzie m oznacza większą wartość z liczby wierzchołków lub liczby krawędzi. Znalezienie w ten sposób minimalnych odległości między wszystkimi parami wierzchołków zajmuje czas $O(m n \log n)$. Jeśli m jest bliskie swojej maksymalnej wartości $m \approx n^2$ to można skorzystać z implementacji algorytmu Dijkstry który działa w czasie $O(n^2)$. Wykonanie go n razy daje czas $O(n^3)$ wykonania algorytmu znajdowania minimalnych odległości między wszystkimi parami wierzchołków.

Istnieje inny algorytm znajdowania minimalnych odległości między wszystkimi parami wierzchołków, noszący nazwę algorytmu Floyda. Jego wykonanie zajmuje czas $O(n^3)$. Operuje na macierzach sąsiedztwa a nie listach sąsiedztwa i jest koncepcyjnie prostszy.

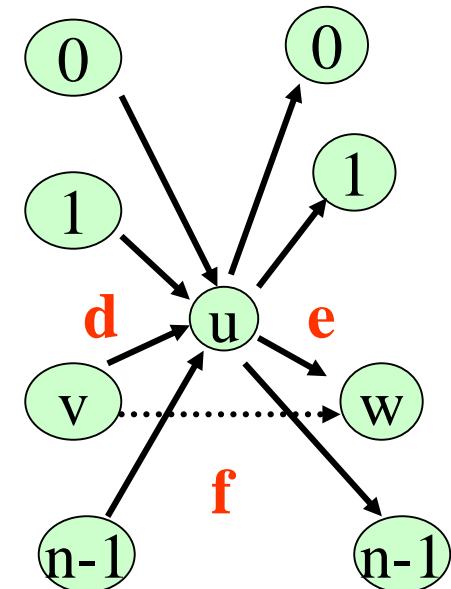
Algorytm Floyda znajdowania najkrótszych dróg

Podstawa algorytmu jest działanie polegające na rozpatrywaniu po kolej ka dego wierzchołka grafu jako elementu centralnego (ang. pivot). Kiedy wierzchołek u jest elementem centralnym staramy się wykorzystać fakt, że u jest wierzchołkiem pośrednim miedzy wszystkimi parami wierzchołków. Dla ka dej pary wierzchołków, na przykład v i w, jeśli suma etykiet krawędzi v → u oraz u → w (na rysunku d + e), jest mniejsza od bieżąco rozpatrywanej etykiety f krawędzi wiod cej od v do w, to warto ć f jest zast opowana wartością d+e.

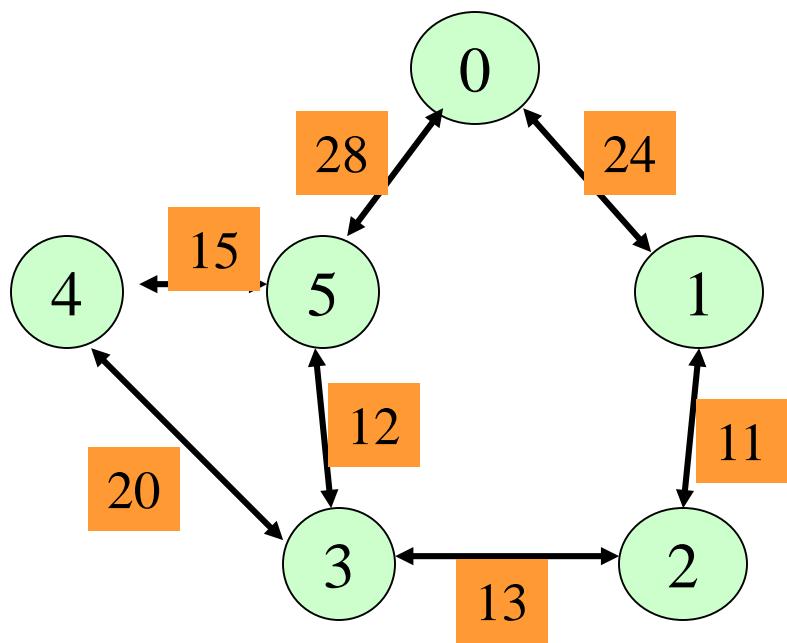
Node u, v, w:

```
for (v = 0; w < MAX; v++)
    for (w=0; w < MAX; w++)
        dist[v][w] = arc[v][w];
for (u=0; v< MAX; v++)
    for (w=0; w<MAX; w++)
        if( dist[v][u]+dist[u][w] < dist[v][w])
            dist[v][w] = dist [v][u] + dist [u][w];
```

$\text{arc}[v][w]$ - etykieta krawędzi, wierzchołki numerowane

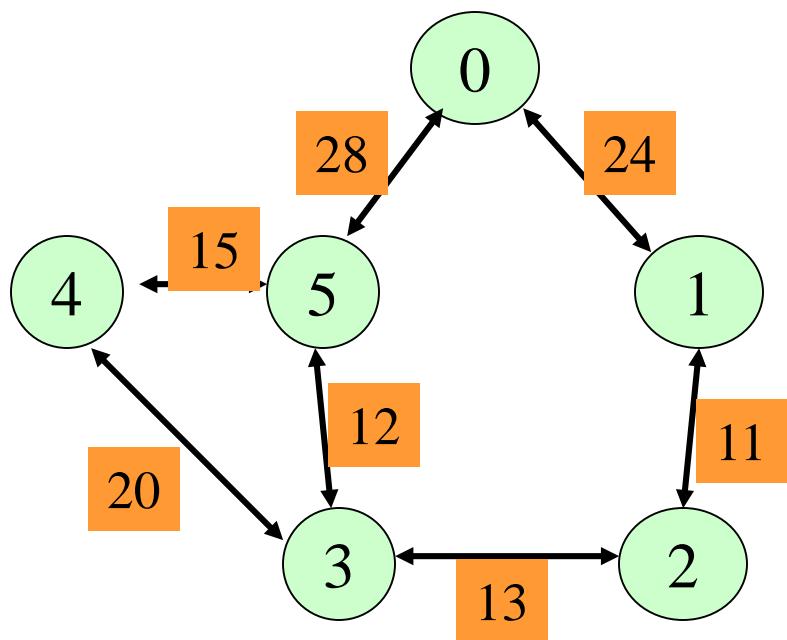


Macierz arc, która odzwierciedla początkową postać macierzy dist



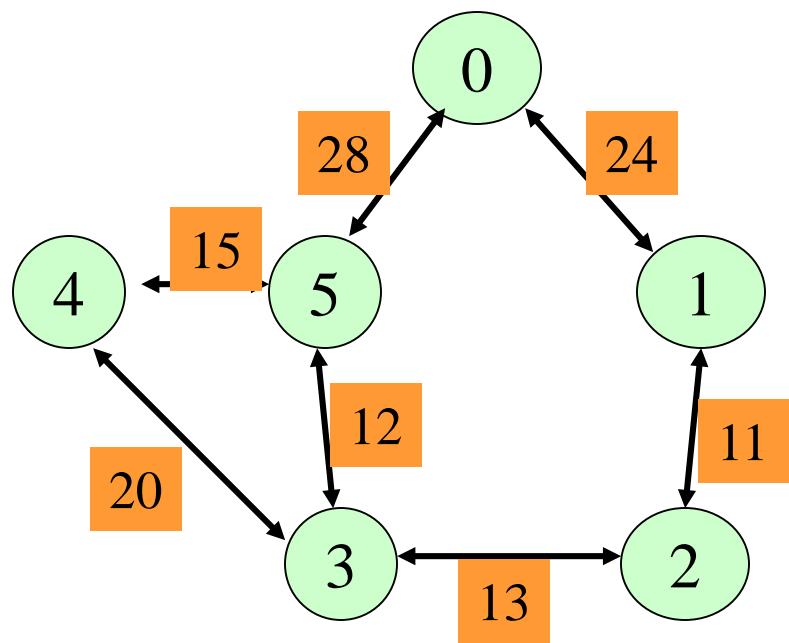
	0	1	2	3	4	5
0	0	24	INF	INF	INF	28
1	24	0	11	INF	INF	INF
2	INF	11	0	13	INF	INF
3	INF	INF	13	0	20	12
4	INF	INF	INF	20	0	15
5	28	INF	INF	12	15	0

Macierz dist, po użyciu wierzchołka
0 jako elementu centralnego



	0	1	2	3	4	5
0	0	24	INF	INF	INF	28
1	24	0	11	INF	INF	52
2	INF	11	0	13	INF	INF
3	INF	INF	13	0	20	12
4	INF	INF	INF	20	0	15
5	28	52	INF	12	15	0

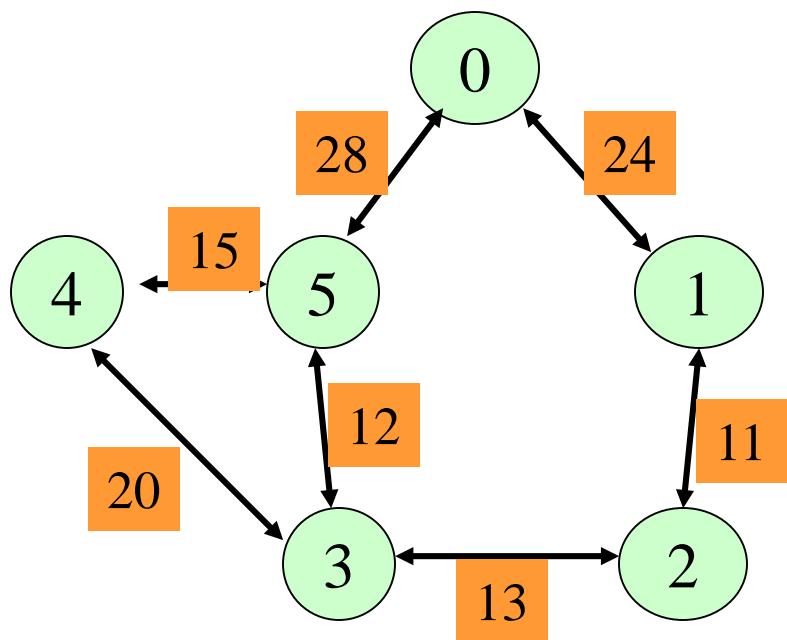
Macierz dist, po użyciu wierzchołka
1 jako elementu centralnego



	0	1	2	3	4	5
0	0	24	35	INF	INF	28
1	24	0	11	INF	INF	52
2	35	11	0	13	INF	63
3	INF	INF	13	0	20	12
4	INF	INF	INF	20	0	15
5	28	52	63	12	15	0

itd.... itd....

Ostateczna postać macierzy dist.



	0	1	2	3	4	5
0	0	24	35	40	43	28
1	24	0	11	24	44	52
2	35	11	0	13	33	25
3	40	24	13	0	20	12
4	43	44	33	20	0	15
5	28	36	25	12	15	0

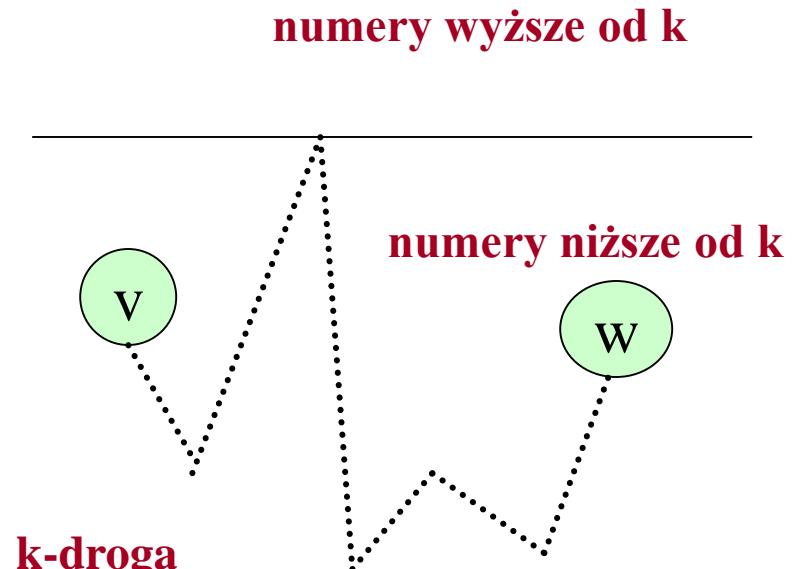
Uzasadnienie poprawności algorytmu Floyda

Na dowolnym etapie działania algorytmu Floyda odległość z wierzchołka v do wierzchołka w stanowi długość najkrótszej z tych dróg, które wiodą jedynie przez wierzchołki użyte dotąd jako elementy centralne. Ponieważ wszystkie wierzchołki zostają w końcu użyte jako elementy centralne, elementy $\text{dist}[v][w]$ zawierają po zakończeniu działań minimalne długości wszystkich możliwych dróg.

Definiujemy k -drogę z wierzchołka v do wierzchołka w jako drogę z v do w taką, że żaden jej wierzchołek pośredni nie ma numeru wyższego od k .

Należy zauważyć, że nie ma ograniczenia odnośnie tego, że v lub w mogą mieć wartość k lub mniejszą.

$k=-1$ oznacza że droga nie posiada wierzchołków pośrednich.



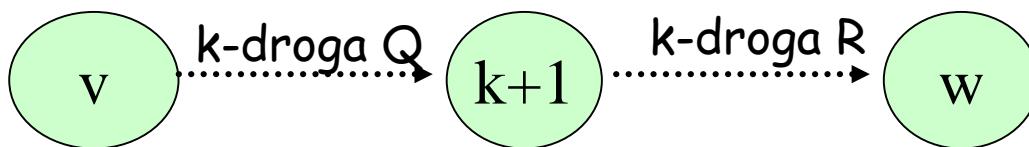
Twierdzenie S(k): jeżeli etykiety krawędzi mają wartości nieujemne, to po przebiegu k - pętli, element $\text{dist}[v][w]$ ma wartość najkrótszej k - drogi z v do w lub ma wartość INF, jeżeli taka droga nie istnieje.

Podstawa:

Podstawa jest warunek $k = -1$. Krawędzie i drogi składające się z pojedynczego wierzchołka są jedynymi (-1) drogami.

Indukcja:

Założymy ze $S(k)$ jest spełnione i rozważmy co się dzieje z elementami $\text{dist}[v][w]$ w czasie $k+1$ przebiegu pętli. Założymy, że P jest najkrótszą $(k+1)$ - drogą wiodącą z v do w. Mamy do czynienia z dwoma przypadkami, w zależności czy droga P prowadzi przez wierzchołek $k+1$.



k-drogę P można rozbić na dwie k-drogi, Q oraz R.

(1) Jeżeli P jest k-drogą, to znaczy, kiedy P nie wiedzie przez wierzchołek $k+1$, to na podstawie hipotezy indukcyjnej wartość elementu $\text{dist}[v][w]$ jest równa długości P po zakończeniu k-tej iteracji. Nie można zmienić wartości $\text{dist}[v][w]$ podczas przebiegu wykonywanego dla wierzchołka $k+1$ traktowanego jako element centralny, gdyż nie istnieją żadne krótsze $(k+1)$ -drogi.

(2) Jeżeli P jest $(k+1)$ -drogą, można założyć, że P przechodzi przez wierzchołek $k+1$ tylko raz, gdyż cykl nigdy nie może spowodować zmniejszenia odległości (przy założeniu że wszystkie etykiety mają wartości nieujemne).

Stąd droga P składa się z k-drogi Q, wiodącej od wierzchołka v do $k+1$, oraz k-drogi R, wiodącej od wierzchołka $k+1$ do w. Na podstawie hipotezy indukcyjnej wartości elementów $\text{dist}[v][k+1]$ oraz $\text{dist}[k+1][w]$ będą długościami dróg odpowiednio, Q i R, po zakończeniu k-tej iteracji.

Ostatecznie wnioskujemy, że w $(k+1)$ przebiegu, wartością elementu $\text{dist}[v][w]$ staje się długość najkrótszej $(k+1)$ -drogi dla wszystkich wierzchołków v oraz w. Jest to twierdzenie $S(k+1)$, co oznacza koniec kroku indukcyjnego.

Założymy teraz, że $k=n-1$. Oznacza to, że wiemy iż po zakończeniu wszystkich n przebiegów, wartość $\text{dist}[v][w]$ będzie minimalną odlegością dowolnej $(n-1)$ -drogi wiodącej z wierzchołka v do w. Ponieważ każda droga jest $(n-1)$ drogą, więc $\text{dist}[v][w]$ jest minimalną długością drogi wiodącej z wierzchołka v do w.

Posumowanie informacji o algorytmach grafowych

PROBLEM	ALGOTYTM(Y)	CZAS WYKONANIA
Minimalne drzewo rozpinające Znajdowanie cykli	Algorytm Kruskala Przeszukiwanie w głąb	$O(m \log n)$ $O(m)$
Uporządkowanie topolog.	Przeszukiwanie w głąb	$O(m)$
Osiagalność w przypadk. pojedynczego źródła	Przeszukiwanie w głąb	$O(m)$
Spójne składowe	Przeszukiwanie w głąb	$O(m)$
Najkrótsza droga dla pojedyncz. źródła	Algorytm Dijkstry	$O(m \log n)$
Najkrótsza droga dla wszystkich par	Algorytm Dijkstry Algorytm Floyda	$O(m n \log n)$ $O(n^3)$

TEORETYCZNE PODSTAWY INFORMATYKI

14/11/2016

WFAiS UJ, Informatyka Stosowana
I rok studiów, I stopień

Wykład 7

2

Modele
danych: grafy

- Podstawowe pojęcia**
- Grafy wywołań**
- Grafy skierowane i nieskierowane**
- Grafy planarne, kolorowanie grafów**
- Implementacja grafów**
 - Listy sąsiedztwa**
 - Macierze sąsiedztwa**
- Składowe spójne, algorytm Kruskala**
- Sortowanie topologiczne**
- Najkrótsze drogi: algorytm Dikstry i Floyda-Warshalla**

Graf

3

- **Graf to jest relacja binarna.**
- **Dla grafów mamy możliwość wizualizacji jako zbiór punktów (zwanych wierzchołkami) połączonych liniami lub strzałkami (nazwanych krawędziami). Pod tym względem graf stanowi uogólnienie drzewiastego modelu danych.**
- **Podobnie jak drzewa, grafy występują w różnych postaciach: grafów skierowanych i nieskierowanych lub etykietowanych i nieetykietowanych.**
- **Grafy są przydatne do analizy szerokiego zakresu problemów: obliczanie odległości, znajdowanie cykliczności w relacjach, reprezentacji struktury programów, reprezentacji relacji binarnych, reprezentacji automatów i układów elektronicznych.**

Podstawowe pojęcia

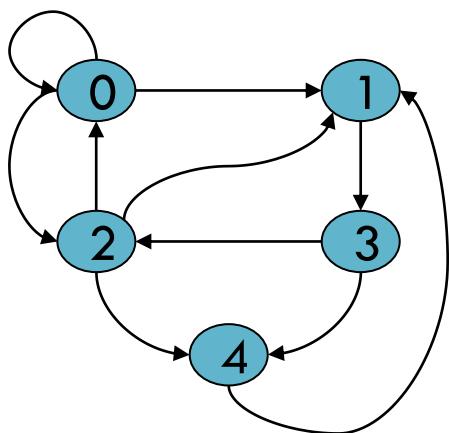
4

□ Graf skierowany (ang. directed graph)

Składa się z następujących elementów:

□ Zbioru **V** wierzchołków (ang. nodes, vertices)

□ Relacji binarnej **E** na zbiorze **V**. Relacje **E** nazywa się zbiorem krawędzi (ang. edges) grafu skierowanego. Krawędzie stanowią zatem pary wierzchołków (**u,v**).



$$V = \{0,1,2,3,4\}$$

$$E = \{ (0,0), (0,1), (0,2), (1,3), (2,0), (2,1), (2,4), (3,2), (3,4), (4,1) \}$$

Podstawowe pojęcia

5

□ Etykiety:

- Podobnie jak dla drzew, dla grafów istnieje możliwość przypisania do każdego wierzchołka etykiety (ang. label).
- Nie należy mylić nazwy wierzchołka z jego etykietą. Nazwy wierzchołków muszą być niepowtarzalne, ale kilka wierzchołków może być oznaczonych ta sama etykieta.



□ Drogi:

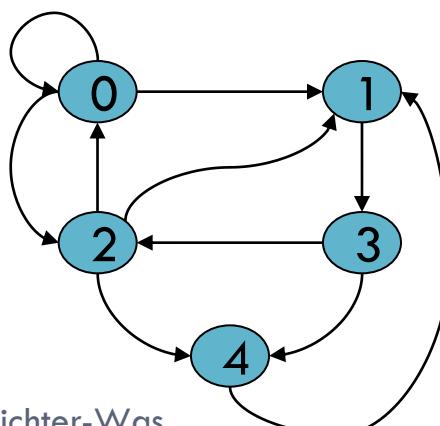
- Droga (ang. path) w grafie skierowanym stanowi listę wierzchołków, (n_1, n_2, \dots, n_k) taka, że występuje krawędź łącząca każdy wierzchołek z następnym, to znaczy $(n_i, n_{i+1}) \in E$ dla $i=1, 2, \dots, k$. Długość (ang. length) drogi wynosi $k-1$, co stanowi liczbę krawędzi należących do tej samej drogi.

Podstawowe pojęcia

6

□ Grafy cykliczne i acykliczne:

- Cykl (ang. cycle) w grafie skierowanym jest drogą o długości 1 lub więcej, która zaczyna się i kończy w tym samym wierzchołku.
- Długość cyklu jest długością drogi. Cykl jest prosty (ang. simple) jeżeli żaden wierzchołek (oprócz pierwszego) nie pojawia się na nim więcej niż raz.
- Jeżeli istnieje cykl nieprosty zawierający wierzchołek n, to można znaleźć prosty cykl który zawiera n. Jeżeli graf posiada jeden lub więcej cykli to mówimy że jest grafem cyklicznym (ang. cyclic). Jeżeli cykle nie występują to, graf określa się mianem acyklicznego (ang. acyclic).



Przykłady cykli prostych:

(0,0), (0,2,0), (1,3,2,1), (1,3,2,4,1)

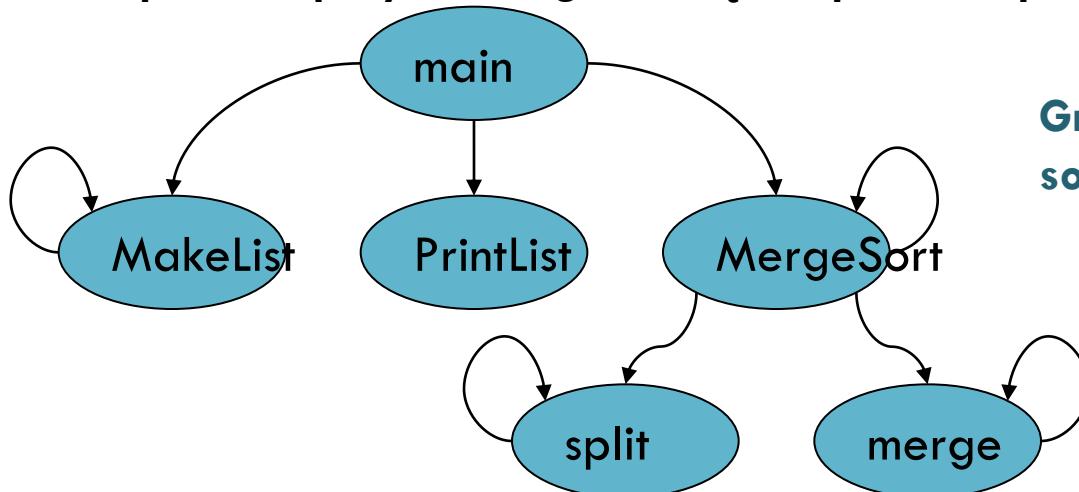
Przykład cyklu nieprostego:

(0,2,1,3,2,0)

Grafy wywołań

7

- Wywołania dokonywane przez zestaw funkcji można reprezentować za pomocą grafu skierowanego, zwanego grafem wywołań. Jego wierzchołki stanowią funkcje, a krawędź (P, Q) istnieje wówczas, gdy funkcja P wywołuje funkcję Q .
- Istnienie cyklu w grafie implikuje występowanie w algorytmie rekurencji.
- Rekurencja w której funkcja wywołuje samą siebie nazywamy bezpośrednią (ang. direct).
- Czasem mamy do czynienia z rekurencją pośrednią (ang. indirect) która reprezentuje cykl o długości większej niż 1, np. (P, Q, R, P) .



Graf wywołań dla algorytmu sortowania przez scalanie

Rekurencja bezpośrednia

Grafy nieskierowane

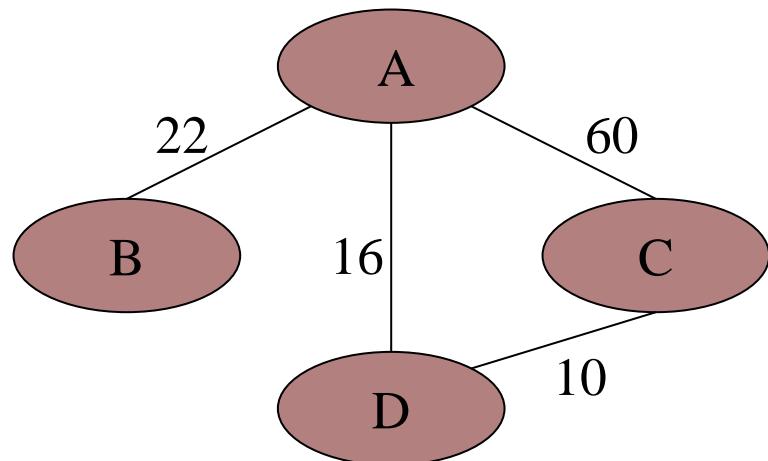
8

- Czasem zasadne jest połączenie wierzchołków krawędziami, które nie posiadają zaznaczonego kierunku. Z formalnego punktu widzenia taka krawędź jest zbiorem dwóch wierzchołków.
- Zapis $\{u,v\}$ mówi ze wierzchołki u oraz v są połączone w dwóch kierunkach. Jeśli $\{u,v\}$ jest krawędzią nieskierowaną, wierzchołki u i v określa się jako **sąsiednie** (ang. *adjacent*) lub mianem **sąsiadów** (ang. *neighbors*).
- Graf zawierający krawędzie nieskierowane, czyli graf z relacją symetryczności krawędzi, nosi nazwę grafu **nieskierowanego** (ang. *undirected graph*).

Grafy nieskierowane

9

- **Droga to lista wierzchołków. Nieco trudniej jest sprecyzować co to jest cykl, tak aby nie była to każda lista**
 $(v_1, v_2, \dots, v_{\kappa-1}, v_\kappa, v_{\kappa-1}, \dots, v_2, v_1)$

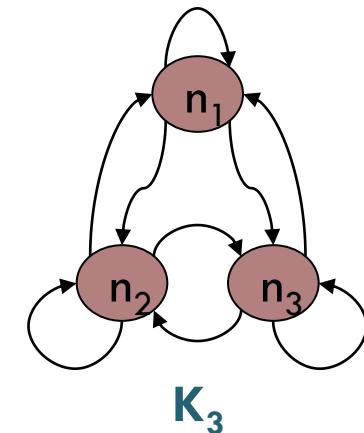
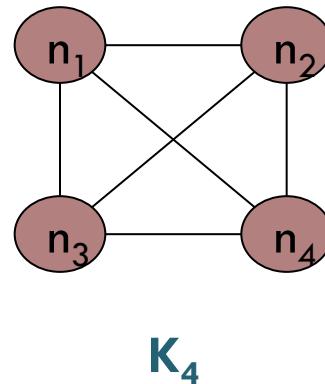
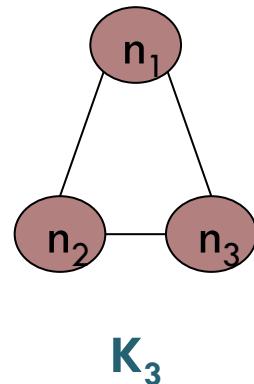
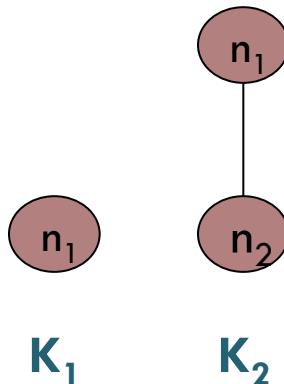


Graf nieskierowany reprezentujący drogi.

Pewne pojęcia z teorii grafów

10

- Teoria grafów jest dziedziną matematyki zajmującą się właściwościami grafów.
- Grafy pełne:
 - Nieskierowany graf posiadający krawędzie pomiędzy każdą parą różnych wierzchołków nosi nazwę grafu pełnego (ang. *complete graph*). Graf pełny o n wierzchołkach oznacza się przez K_n .
 - Liczba krawędzi w nieskierowanym grafie K_n wynosi $n(n-1)/2$, w skierowanym grafie K_n wynosi n^2 .

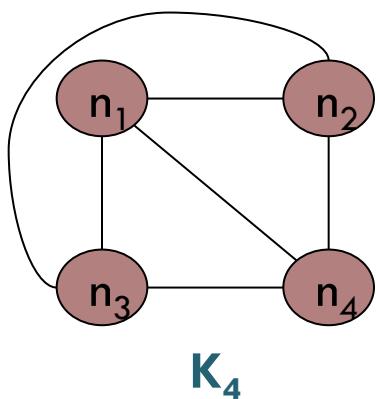


Grafy planarne i nieplanarne

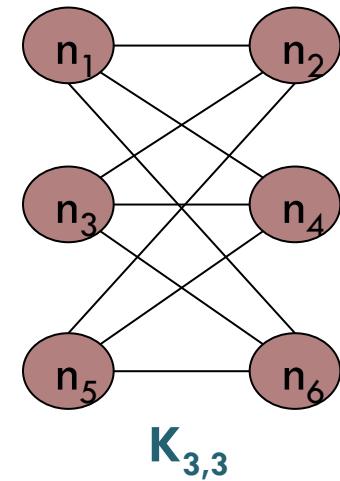
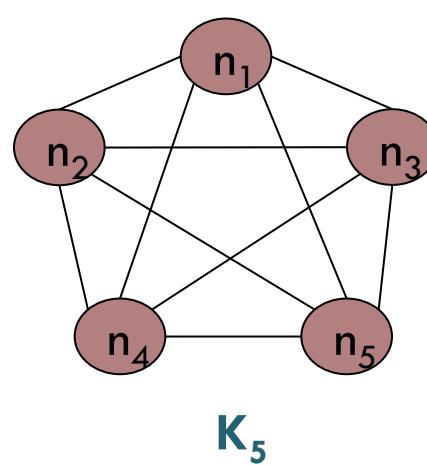
11

- O grafie nieskierowanym mówi się że jest **planarny** (ang. **planar**) wówczas, gdy istnieje możliwość rozmieszczenia jego wierzchołków na płaszczyźnie, a następnie narysowania jego krawędzi jako lini ciągłych które się nie przecinają.
- **Grafe nieplanarne** (ang. **nonplanar**) to takie które nie posiadają reprezentacji płaskiej.

Reprezentacja planarna:



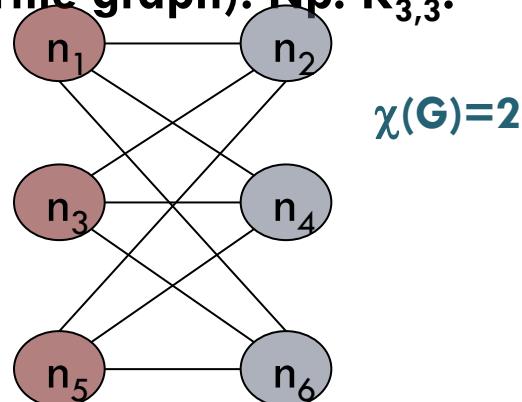
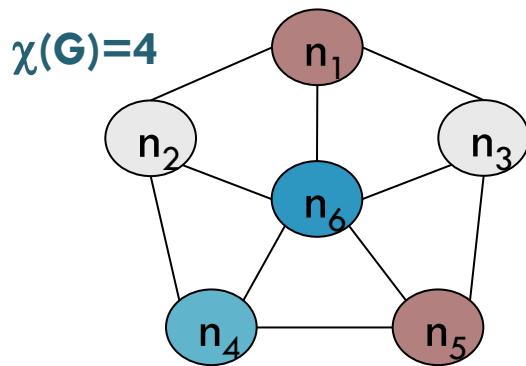
Najprostsze grafy nieplanarne:



Kolorowanie grafów

12

- Kolorowanie grafu (ang. graph coloring) polega na przypisaniu do każdego wierzchołka pewnego koloru, tak aby żadne dwa wierzchołki połączone krawędzią nie miały tego samego koloru.
- Minimalna liczba kolorów potrzebna do takiej operacji nazwana jest liczbą chromatyczną grafu (ang. chromatic number), oznaczaną $c(G)$.
 - Jeżeli graf jest pełny to jego liczba chromatyczna jest równa liczbie wierzchołków
 - Jeżeli graf możemy pokolorować przy pomocy dwóch kolorów to nazywamy go dwudzielnym (ang. bipartite graph). Np. $K_{3,3}$.



Implementacje grafów

13

- Istnieją dwie standardowe metody reprezentacji grafów:
 - Listy sąsiedztwa (ang. *adjacency lists*), jest, ogólnie rzecz biorąc, podobna do implementacji relacji binarnych.
 - Macierze sąsiedztwa (ang. *adjacency matrices*), to nowy sposób reprezentowania relacji binarnych, który jest bardziej odpowiedni dla relacji, w przypadku którym liczba istniejących par stanowi znaczącą część całkowitej liczby par, jakie mogłyby teoretycznie istnieć w danej dziedzinie.

Listy sąsiedztwa

14

- Wierzchołki są ponumerowane kolejnymi liczbami całkowitymi **0,1,....., MAX-1** lub oznaczone za pomocą innego adekwatnego typu wyliczeniowego (używamy poniżej typu **NODE** jako synonimy typu wyliczeniowego).
- Wówczas można skorzystać z podejścia opartego na wektorze własnym.
- Element **successors[u]** zawiera wskaźnik do listy jednokierunkowej wszystkich bezpośrednich następców wierzchołka u. Następniki mogą występować w dowolnej kolejności na liście jednokierunkowej.

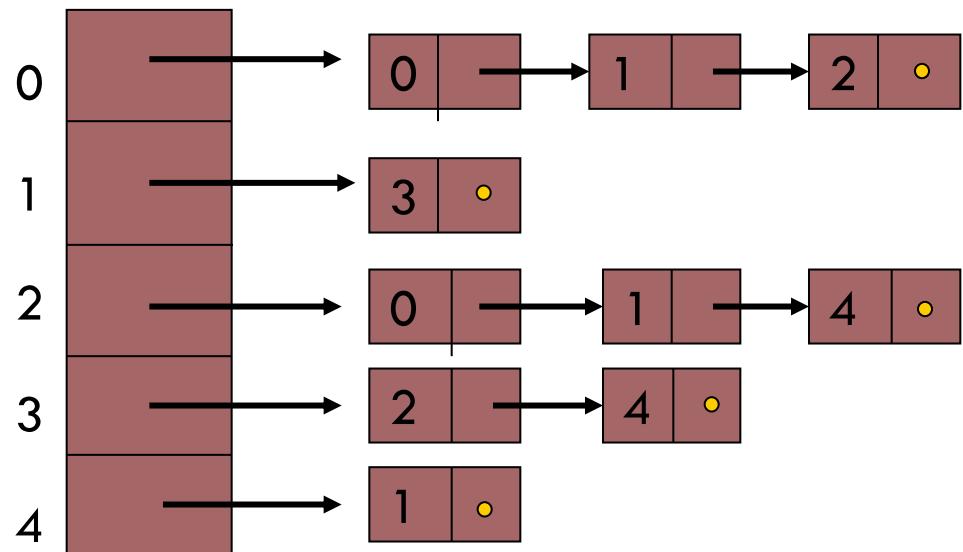
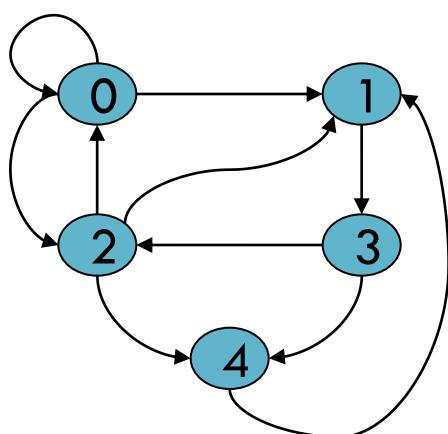
Listy sąsiedztwa:

```
typedef struct CELL *LIST;
struct CELL {
    NODE nodeName;
    LIST next;
}
LIST successors[MAX]
```

Listy sąsiedztwa

15

- Listy sąsiedztwa zostały posortowane wg. kolejności, ale następcy mogą występować w dowolnej kolejności na odpowiedniej liście sąsiedztwa.



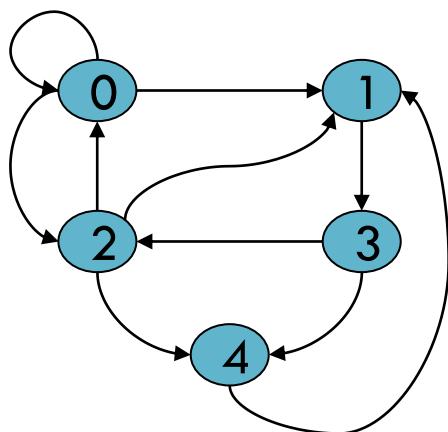
Macierz sąsiedztwa

16

Tworzymy dwuwymiarową tablicę;

BOOLEAN vertices[MAX][MAX];

w której element **vertices[u][v]** ma wartość **TRUE** wówczas, gdy istnieje krawędź (u, v), zaś **FALSE**, w przeciwnym przypadku.



	0	1	2	3	4
0	1	1	1	0	0
1	0	0	0	1	0
2	1	1	0	0	1
3	0	0	1	0	1
4	0	1	0	0	0

Listy sąsiedztwa a macierz sąsiedztwa

17

- **Macierze sąsiedztwa** są preferowanym sposobem reprezentacji grafów wówczas, gdy **grafy są gęste** (ang. dense), to znaczy, kiedy liczba krawędzi jest bliska maksymalnej możliwej ich liczby.
- Dla grafu skierowanego o n wierzchołkach maksymalna liczba krawędzi wynosi n^2 .
- Jeśli **graf jest rzadki** (ang. sparse) to reprezentacja oparta na **listach sąsiedztwa** może pozwolić zaoszczędzić pamięć.
- Istotne różnice między przedstawionymi reprezentacjami grafów są widoczne już przy wykonywaniu prostych operacji.

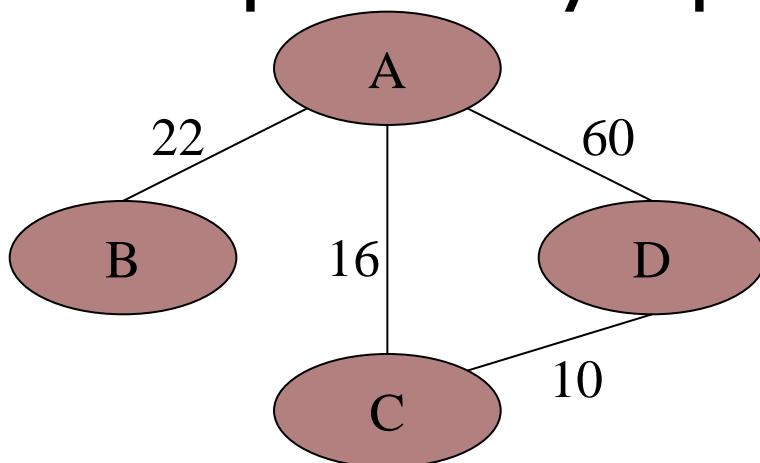
Preferowany sposób reprezentacji:

OPERACJA	GRAF GESTY	GRAF RZADKI
Wyszukiwanie krawędzi	Macierz sąsiedztwa	Obie
Znajdowanie następców	Obie	Lista sąsiedztwa
Znajdowanie poprzedników	Macierz sąsiedztwa	Obie

Składowa spójna grafu nieskierowanego

18

- Każdy graf nieskierowany można podzielić na jedną lub większą liczbę spójnych składowych (ang. connected components).
- Każda spójna składowa to taki zbiór wierzchołków, że dla każdych dwóch z tych wierzchołków istnieje łącząca je ścieżka. Jeżeli graf składa się z jednej spójnej składowej to mówimy że jest spójny (ang. connected).



To jest graf spójny

Algorytm wyznaczania spójnych składowych

19

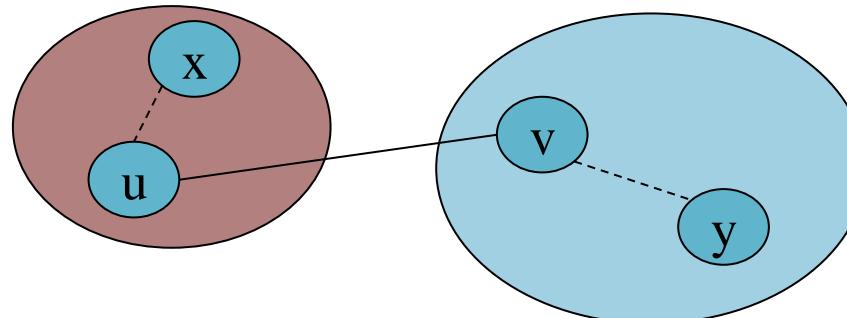
Chcemy określić spójne składowe grafu G. Przeprowadzamy rozumowanie indukcyjne.

Podstawa:

- **Graf G_0 zawiera jedynie wierzchołki grafu G i żadnej jego krawędzi. Każdy wierzchołek stanowi odrębną spójną składową .**

Indukcja:

- **Zakładamy, że znamy już spójne składowe grafu G, po rozpatrzeniu pierwszych i krawędzi, a obecnie rozpatrujemy (i+1) krawędź $\{u, v\}$.**
 - jeżeli wierzchołki u, v należą do jednej spójnej składowej to nic się nie zmienia
 - jeżeli do dwóch różnych, to łączymy te dwie spójne składowe w jedną.



Struktura danych dla wyznaczania spójnych składowych

20

- Biorąc pod uwagę przedstawiony algorytm, musimy zapewnić szybką wykonywalność następujących operacji:
 - 1) gdy jest określony wierzchołek to znajdź jego bieżącą spójną składową
 - 2) połącz dwie spójne składowe w jedną
- Dobre wyniki daje ustawienie **wierzchołków każdej składowej w strukturze drzewiastej**, gdzie spójna składowa jest reprezentowana przez korzeń.
 - aby wykonać operacje (1) należy przejść do korzenia: $O(\log n)$
 - aby wykonać operacje (2) wystarczy korzeń jednej składowej określić jako potomka korzenia składowej drugiej ($O(1)$) .
 - Przyjmijmy zasadę ze korzeń drzewa o mniejszej wysokości czynimy potomkiem.
- Wyznaczenie wszystkich spójnych składowych $O(m \log n)$, m - krawędzi i n - wierzchołków.

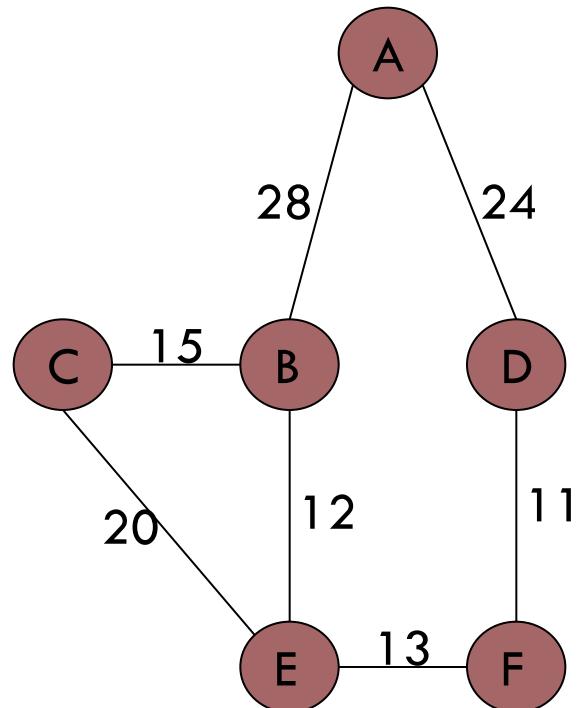
Minimalne drzewa rozpinające

21

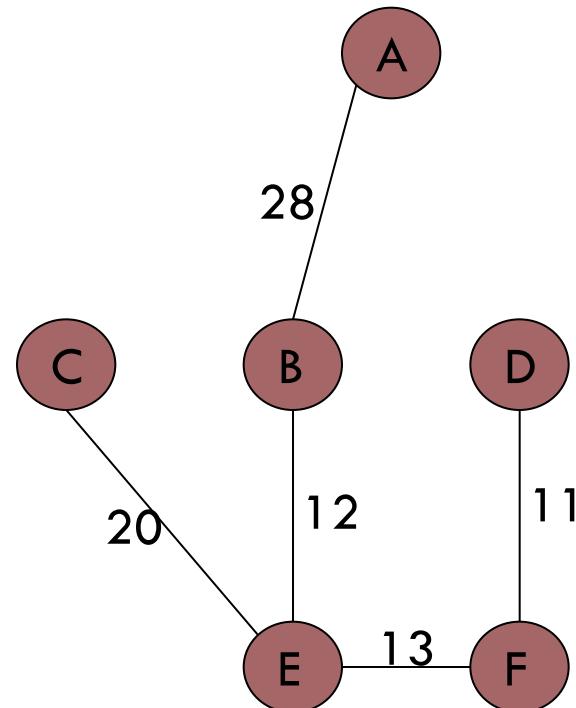
- Drzewo rozpinające (ang. *spanning tree*) grafu nieskierowanego G stanowi zbiór wierzchołków tego grafu wraz z podzbiorem jego krawędzi, takich że:
 - łączą one wszystkie wierzchołki, czyli istnieje droga między dwoma dowolnymi wierzchołkami która składa się tylko z krawędzi drzewa rozpinającego.
 - tworzą one drzewo nie posiadające korzenia, nieuporządkowane. Oznacza to że nie istnieją żadne (proste) cykle.
- Jeśli graf G stanowi pojedynczą spójną składową to drzewo rozpinające zawsze istnieje. Minimalne drzewo rozpinające (ang. *minimal spanning tree*) to drzewo rozpinające, w którym suma etykiet jego krawędzi jest najmniejsza ze wszystkich możliwych do utworzenia drzew rozpinających tego grafu.

Minimalne drzewa rozpinające

22



Graf nieskierowany



Drzewo rozpinające

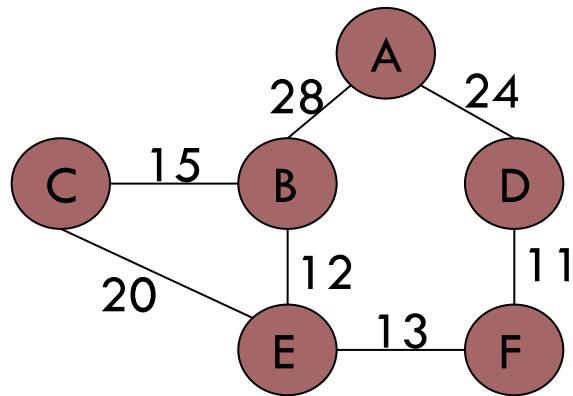
Algorytm Kruskala

23

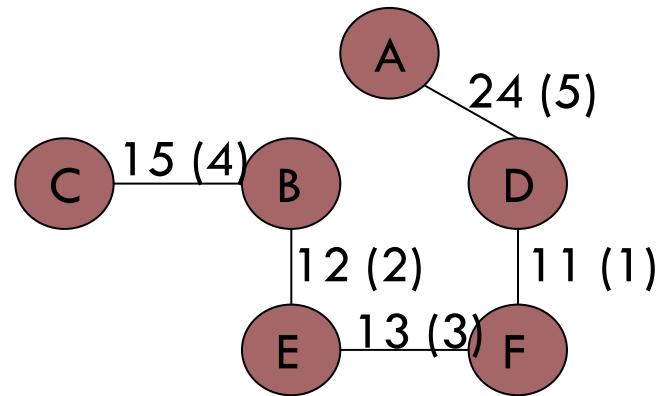
- Istnieje wiele algorytmów do znajdowania minimalnego drzewa rozpinającego.
- Jeden z nich to algorytm Kruskala, który stanowi proste rozszerzenie algorytmu znajdowania spójnych składowych. Wymagane zmiany to:
 - należy rozpatrywać krawędzie w kolejności zgodnej z rosnącą wartością ich etykiet,
 - należy dołączyć krawędź do drzewa rozpinającego tylko w takim wypadku gdy jej końce należą do dwóch różnych spójnych składowych.

Algorytm Kruskala

24



Graf nieskierowany



Minimalne drzewo rozpinające
(w nawiasach podano kolejność dodawanych krawędzi)

Algorytm Kruskala

25

- **Algorytm Kruskala jest dobrym przykładem algorytmu zachłanego (ang. greedy algorithm), w przypadku którego podejmowany jest szereg decyzji, z których każdą stanowi wybór opcji najlepszej w danym momencie.**
 - Lokalnie podejmowane decyzje polegają w tym przypadku na wyborze krawędzi dodawanej do formowanego drzewa rozpinającego.
 - Za każdym razem wybierana jest krawędź o najmniejsze wartości etykiety, która nie narusza definicji drzewa rozpinającego, zabraniającej utworzenia cyklu.
- **Dla algorytmu Kruskala można wykazać, że jego rezultat jest optymalny globalnie, to znaczy że daje on w wyniku drzewo rozpinające o minimalnej wadze.**
- **Czas wykonania algorytmu jest $O(m \log m)$ gdzie m to jest większa z wartości liczby wierzchołków i liczby krawędzi.**

Uzasadnienie poprawności algorytmu Kruskala

26

- **Niech G będzie nieskierowanym grafem spójnym.**
(Dla niektórych etykiet dopuszczałyśmy dodanie nieskończoności malej wartości tak aby wszystkie etykiety były różne, graf G będzie miał wobec tego unikatowe minimalne drzewo rozpinające, które będzie jednym spośród minimalnych drzew rozpinających grafu G_0 oryginalnych wagach).
- **Niech ciąg e_1, e_2, \dots, e_m oznacza wszystkie krawędzie grafu G w kolejności zgodnej z rosnącą wartością ich etykiet, rozpoczynając od najmniejszej.**
- **Niech K będzie drzewem rozpinającym grafu G_0 odpowiednio zmodyfikowanych etykietach, utworzonym przez zastosowanie algorytmu Kruskala, a T niech będzie unikatowym minimalnym drzewem rozpinającym grafu G .**

Uzasadnienie poprawności algorytmu Kruskala

27

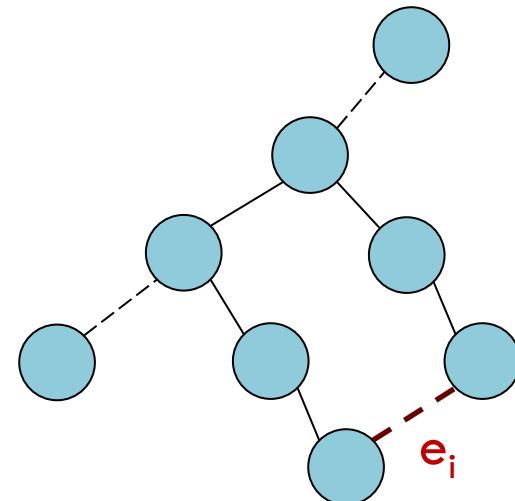
- Należy udowodnić ze K i T stanowią to samo drzewo. Jeśli są różne musi istnieć co najmniej jedna krawędź, która należy do jednego z nich a nie należy do drugiego.
- Niech e_i oznacza pierwszą taką krawędź spośród uporządkowanych krawędzi, to znaczy każda z krawędzi e_1, e_2, \dots, e_{i-1} albo należy do obu drzew K i T albo nie należy do żadnego z nich.
- Istnieją dwa przypadki w zależności czy krawędź e_i należy do drzewa K czy do drzewa T . W każdym z tych przypadków wykażemy sprzeczność, co będzie stanowić dowód, że e_i nie może istnieć, a stąd że $K=T$, oraz że K stanowi minimalne drzewo rozpinające grafu G .

Uzasadnienie poprawności algorytmu Kruskala

28

□ Przypadek 1:

- krawędź e_i należy do T , ale nie należy do K .
- Jeżeli algorytm Kruskala odrzuca e_i , oznacza to że e_i formuje cykl z pewna droga P , utworzoną z uprzednio wybranych krawędzi drzewa K .
- Jeżeli krawędzie drogi P należą do K to należą także do T .
- A więc $P + e_i$ utworzyłoby cykl w T co jest sprzeczne z definicją drzewa rozpinającego. Stąd niemożliwe jest aby e_i należała do T a nie należała do K .



Droga P (linia ciągła) należy zarówno do drzewa T jak i K , krawędź e_i należy tylko do T .

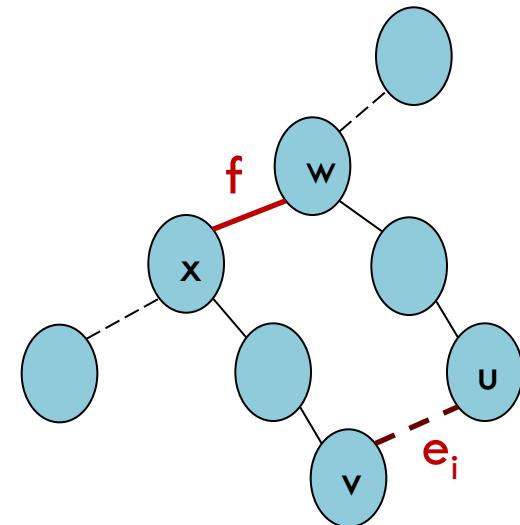
Uzasadnienie poprawności algorytmu Kruskala

29

□ Przypadek 2:

■ krawędź e_i należy do K , ale nie należy do T .
Niech krawędź e_i łączy wierzchołki u i v .
Ponieważ drzewo T jest spójne, musi istnieć
w T pewna acykliczna droga z wierzchołka u
do v . Niech nosi ona nazwę Q . Ponieważ w
skład Q nie wchodzi e_i , $Q + e_i$ tworzy cykl
prosty w grafie G . Rozpatrzmy dwie sytuacje:

- krawędź e_i posiada najwyższą wartość etykiety. Musiałoby to oznaczać że K zawiera cykl co jest niemożliwe.
- na drodze Q istnieje krawędź f która ma wartość etykiety wyższą niż e_i . Można by wiec usunąć f a wprowadzić e_i nie niszcząc spójności. A wiec rozpięte drzewo miałoby wartość mniejszą niż wartość dla T co jest w sprzeczności z początkowym twierdzeniem że T jest minimalne.



Droga Q (linia ciągła)
należy do drzewa T ,
można dodać krawędź
 e_i i usunąć krawędź f

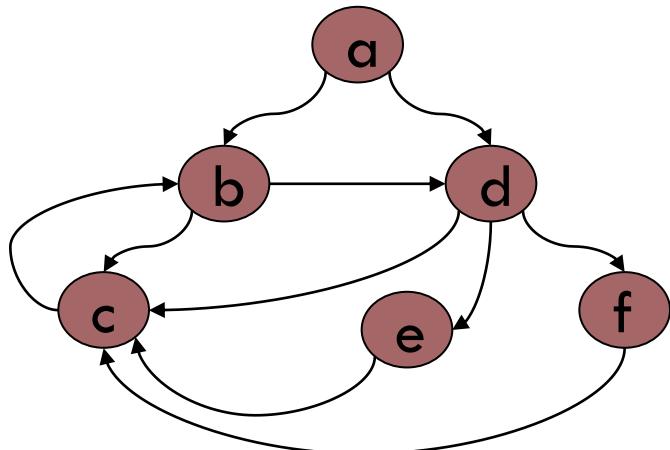
Algorytm przeszukiwania w głąb

30

- Jest to podstawowa metoda badania grafów skierowanych.
- Bardzo podobna do stosowanych dla drzew, w których startuje się od korzenia i rekurencyjnie bada wierzchołki potomne każdego odwiedzonego wierzchołka.
- Trudność polega na tym ze w grafie mogą pojawiać się cykle... Należy wobec tego znaczyć wierzchołki już odwiedzone i nie wracać więcej do takich wierzchołków.
 - Z uwagi na fakt, że w celu uniknięcia dwukrotnego odwiedzenia tego samego wierzchołka jest on odpowiednio oznaczany, graf w trakcie jego badania zachowuje się podobnie do drzewa.
- W rzeczywistości można narysować drzewo, którego krawędzie rodzic-potomek będą niektórymi krawędziami przeszukiwanego grafu G.
 - Takie drzewo nosi nazwę **drzewa przeszukiwania w głąb** (ang. depth-first-search) dla danego grafu.

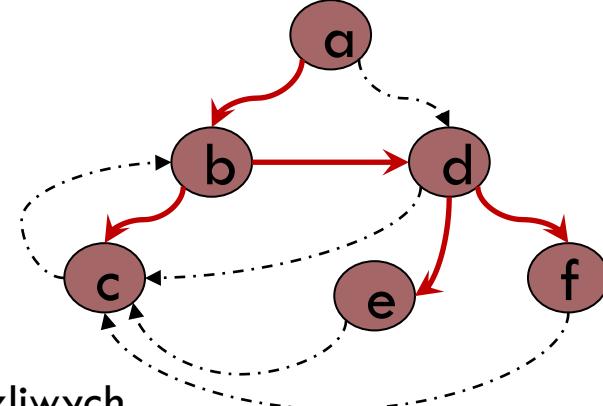
Drzewa przeszukiwania w głąb

31

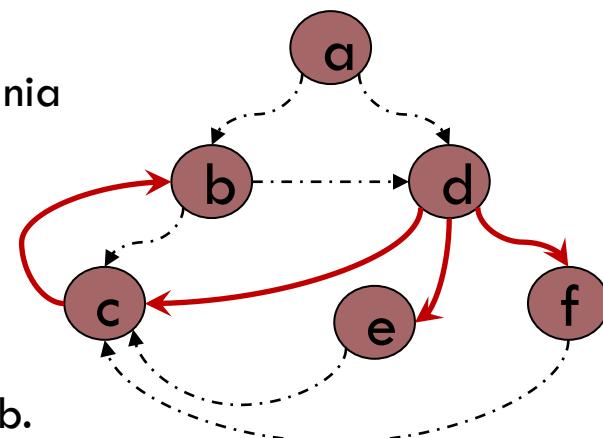


Graf skierowany

Las przeszukiwania:
dwa drzewa o korzeniach a, d



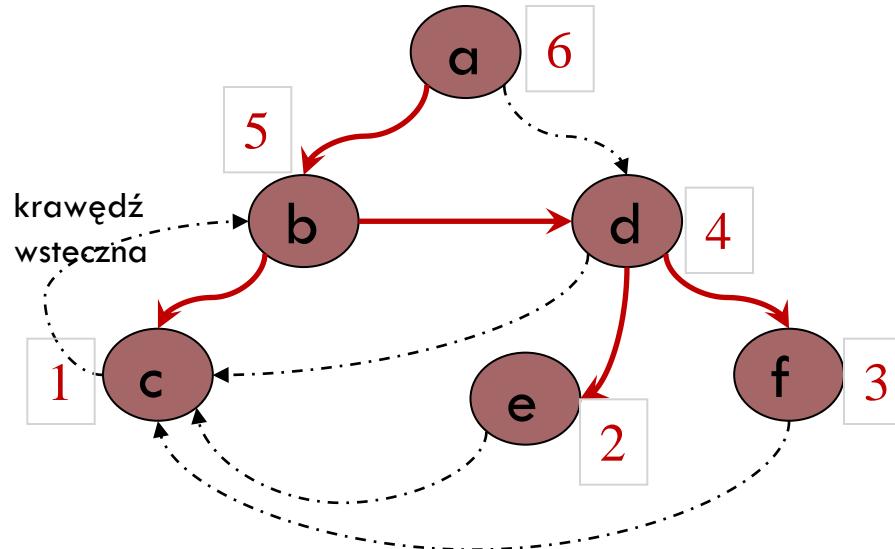
Jedno z możliwych
drzew
przeszukiwania



Las przeszukiwania w głąb.

Drzewo przeszukiwania w głąb

32



- Po (podczas) konstruowaniu drzewa przeszukiwania w głąb można ponumerować jego wierzchołki w **kolejności wstecznej** (ang. *post-order*).

Cykle w grafie skierowanym

33

- Podczas przeszukiwania w głąb grafu skierowanego G można wszystkim wierzchołkom przypisać numery zgodne z kolejnością wsteczną w czasie rzędu $O(m)$.
- Krawędzie wsteczne to takie dla których początki są równe lub mniejsze końcom ze względu na numerację wsteczną.
Zawsze gdy istnieje krawędź wsteczna w grafie musi istnieć cykl.
- Prawdziwe jest również twierdzenie odwrotne. Aby stwierdzić czy w grafie występuje cykl należy przeprowadzić numerację wsteczną a następnie sprawdzić wszystkie krawędzie.
- Całkowity czas wykonania testu cykliczności to $O(m)$, gdzie m to większa z wartości liczby wierzchołków i liczby krawędzi.

Sortowanie topologiczne

34

- Założmy, że graf skierowany G jest acykliczny.
- Dla każdego grafu możemy określić las poszukiwania w głębi, określając numerację wsteczną jego wierzchołków.
- Założmy, że (n_1, n_2, \dots, n_k) określa listę wierzchołków grafu G w kolejności odwrotnej do numeracji wstecznej. To znaczy: n_1 jest wierzchołkiem opatrzonym numerem n , n_2 wierzchołkiem opatrzonym numerem $n-1$ i ogólnie wierzchołek n_i jest opatrzony numerem $n-i+1$.
- Kolejność wierzchołków na tej liście ma ta własność, że wszystkie krawędzie grafu G biegą od początku do końca, tzn. początek poprzedza koniec.

Sortowanie topologiczne

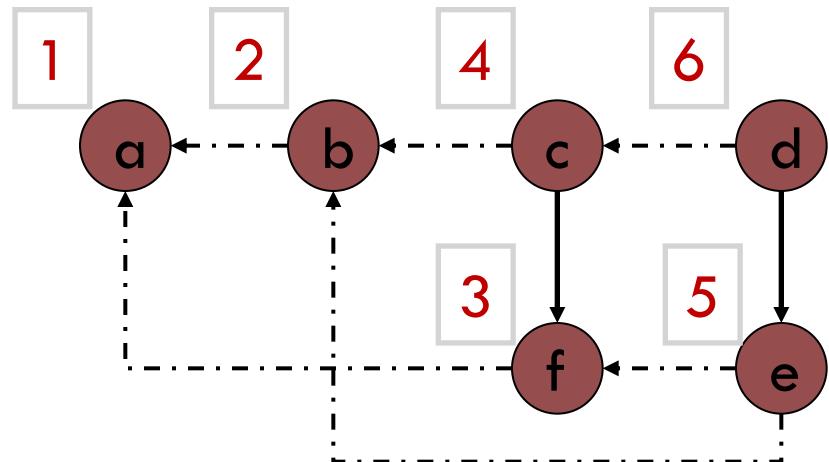
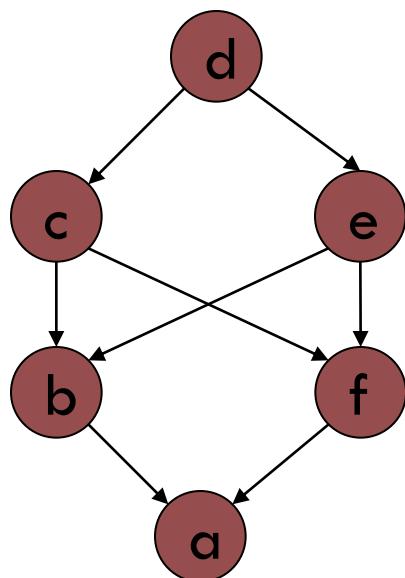
35

- Takie uporządkowanie nazywamy **topologicznym** (ang. *topological order*), a proces znajdowania takiego uporządkowania to **sortowanie topologiczne** (ang. *topological sorting*).
- Jedynie grafy acykliczne posiadają uporządkowanie topologiczne.
- Wykonując poszukiwanie w głąb możemy je określić w czasie **$O(m)$** .
- Jedna z możliwości: odkładać kolejno znalezione wierzchołki „na stos”. Po zakończeniu lista znajdująca się na stosie będzie reprezentować uporządkowanie topologiczne grafu.

Sortowanie topologiczne

36

Uporządkowanie topologiczne to (d, e, c, f, b, a)



Las przeszukiwania w głąb

Skierowany graf cykliczny

Sortowanie topologiczne

37

- **Uporządkowanie topologiczne przydaje się wówczas, gdy istnieją pewne ograniczenia odnośnie kolejności w jakiej mają być wykonywane zadania.**
 - **Jeśli krawędź wiodąca od wierzchołka u do wierzchołka v jest rysowana wówczas, gdy zadanie u musi zostać wykonane przed zadaniem v , to uporządkowaniem zapewniającym wykonanie wszystkich zadań jest właśnie uporządkowanie topologiczne.**

Sortowanie topologiczne

38

- Podobny przykład to **graf wywołań** nierekurencyjnego zbioru funkcji, kiedy należy przeanalizować każdą funkcję dopiero po dokonaniu analizy funkcji ją wywołujączej.
 - Jeśli krawędzie wiodą od funkcji wywołujących do wywoływanych, kolejność, w której należy przeprowadzić takie analizy, to **odwrócenie porządku topologicznego**, czyli **uporządkowanie wsteczne**.
 - Zapewnia to że każda funkcja zostanie przeanalizowana dopiero po dokonaniu analizy wszystkich innych wywoływanych przez nią funkcji.

Sortowanie topologiczne

39

- **Istnienie cyklu** w grafie reprezentującym priorytety zadań mówi o tym, że nie istnieje takie uporządkowanie, dzięki któremu możliwe byłoby wykonanie wszystkich zadań.
- **Istnienie cyklu** w grafie wywołań pozwala stwierdzić występowanie rekurencji.

Problem osiągalności

40

- Naturalne pytanie związane z grafem skierowanym jest:
 - które wierzchołki są osiągalne z danego wierzchołka u przy założeniu, że po grafie można się poruszać tylko zgodnie z kierunkiem krawędzi?
Taki zbiór wierzchołków określa się mianem **zbioru osiągalności** (ang. *reachable set*) danego wierzchołka u.
- Możemy wykorzystać rekurencyjną funkcję poszukiwania w głąb. Całkowity czas wykonania takiego zapytania to **$O(m n)$** .

Znajdowanie spójnych składowych

41

- Do znajdowania spójnych składowych możemy użyć algorytmu poszukiwania w głąb.
- Traktujemy graf nieskierowany jako graf skierowany, w którym każda krawędź nieskierowana została zastąpiona dwiema krawędziami skierowanymi wiodącymi w obu kierunkach.
- Do reprezentacji grafu używamy list sąsiedztwa.
- Tworzymy las przeszukiwania w głąb grafu skierowanego. Każde drzewo w tym lesie odpowiada jednej składowej spójności grafu nieskierowanego.
- Czas wykonania algorytmu $O(m)$
 - ▣ przy użyciu struktury drzewiastej czas wykonania wynosi $O(m \log n)$.

Algorytm Dikstry

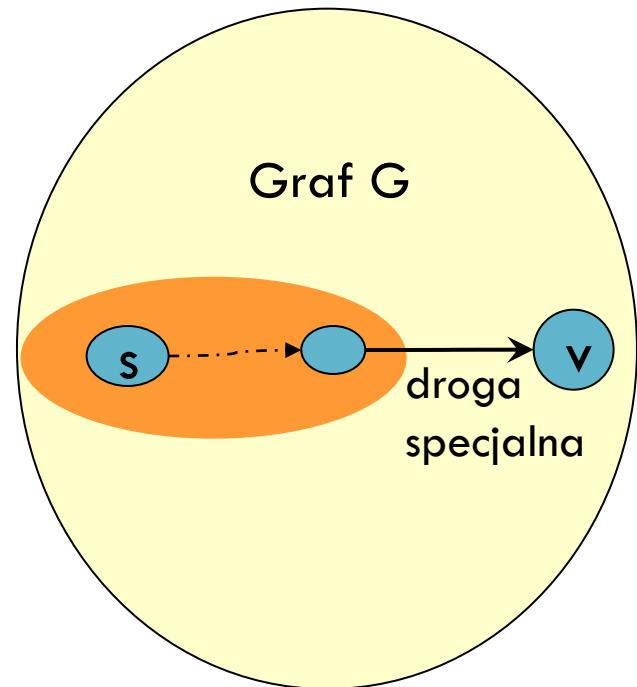
42

- Szukamy najkrótszej drogi pomiędzy dwoma wierzchołkami
 - Rozpatrujemy graf G (skierowany lub nieskierowany), w którym wszystkie krawędzie zaetykietowano wartościami reprezentującymi ich długości.
 - Długość (ang. distance) danej drogi stanowi wartość sumy etykiet związanych z nią krawędzi. Minimalna odległość z wierzchołka u do wierzchołka v to minimalna długość którejś z dróg od u do v .

Algorytm Dikstry

43

- Traktujemy wierzchołek **s** jako wierzchołek źródłowy. Na etapie pośrednim wykonywania algorytmu w grafie G istnieją tzw. wierzchołki ustalone (ang. settled), tzn. takie dla których znane są odległości minimalne. W szczególności zbiór takich wierzchołków zawiera również wierzchołek **s**.
- Dla nieustalonego wierzchołka **v** należy zapamiętać długość najkrótszej drogi specjalnej (ang. special path) czyli takiej która rozpoczyna się w wierzchołku źródłowym, wiedzie przez ustalone wierzchołki, i na ostatnim etapie przechodzi z obszaru ustalonego do wierzchołka **v**.



Algorytm Dikstry

44

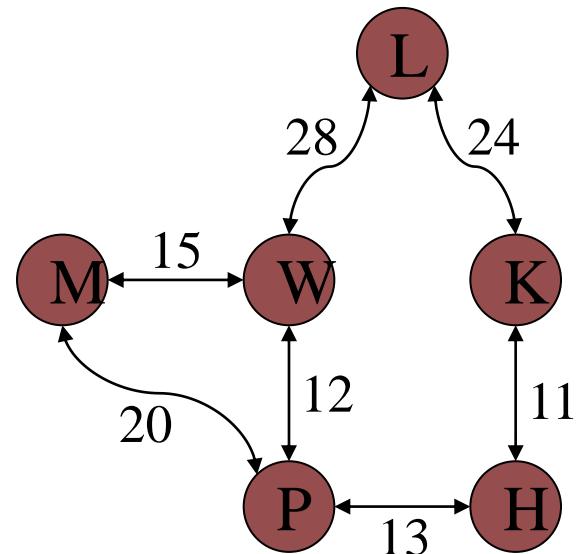
- Dla każdego wierzchołka u zapamiętujemy wartość $\text{dist}(u)$.
- Jeśli u jest wierzchołkiem ustalonym, to $\text{dist}(u)$ jest długością najkrótszej drogi ze źródła do wierzchołka u .
- Jeśli u nie jest wierzchołkiem ustalonym, to $\text{dist}(u)$ jest długością drogi specjalnej ze źródła do u .
- Na czym polega ustalanie wierzchołków:
 - znajdujemy wierzchołek v który jest nieustalony ale posiada najmniejszą $\text{dist}(v)$ ze wszystkich wierzchołków nieustalonych
 - przyjmujemy wartość $\text{dist}(v)$ za minimalną odległość z s do v
 - dostosowujemy wartości wszystkich $\text{dist}(u)$ dla innych wierzchołków, które nie są ustalone, wykorzystując fakt, że wierzchołek v jest już ustalony.
 - Czyli porównujemy stare $\text{dist}(u)$ z wartością $\text{dist}(v) + \text{etykieta}(v, u)$ jeżeli taka (v, u) krawędź istnieje.
- Czas wykonania algorytmu jest $O(m \log n)$.

Algorytm Dikstry

45

Etapy wykonania algorytmu

MIASTO	ETAPY ustalania wierzchołków				
	(1)	(2)	(3)	(4)	(5)
H	0*	0*	0*	0*	0*
P	13	13	13*	13*	13*
M	INF	INF	33	33	33*
W	INF	INF	25	25*	25*
L	INF	35	35	35	35
K	11	11*	11*	11*	11*



Indukcyjny dowód poprawności algorytmu Dikstry

46

- W celu wykazania poprawności algorytmu Dijkstry należy przyjąć, że etykiety krawędzi są nieujemne.
- Indukcyjny dowód poprawności względem k prowadzi do stwierdzenia że:
 - 1) dla każdego wierzchołka ustalonego u , wartość $\text{dist}(u)$ jest minimalną odlegością z s do u , a najkrótsza droga do u składa się tylko z wierzchołków ustalonych.
 - 2) dla każdego nieustalonego wierzchołka u , wartość $\text{dist}(u)$ jest minimalną długością drogi specjalnej z s do u (jeśli droga nie istnieje wartość wynosi INF).

Indukcyjny dowód poprawności algorytmu Dikstry

47

□ Podstawa:

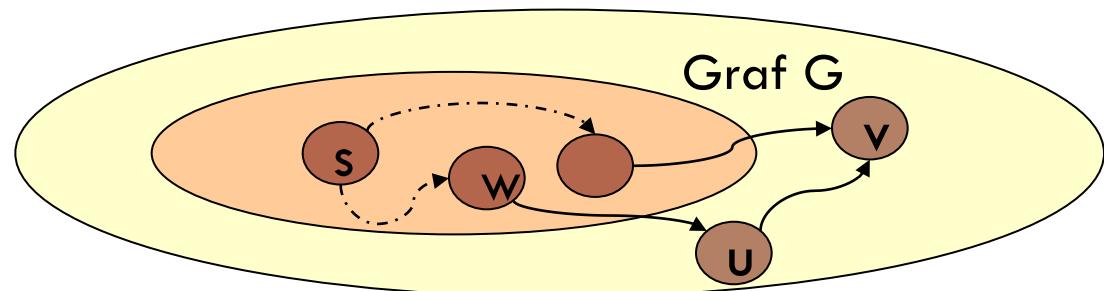
- Dla $k=1$ wierzchołek s jest jedynym wierzchołkiem ustalonym. Inicjalizujemy $\text{dist}(s)$ wartością 0, co spełnia warunek (1).
- Dla każdego innego wierzchołka u , $\text{dist}(u)$ jest inicjalizowane wartością etykiety krawędzi (s, u) , o ile taka istnieje. Jeżeli nie istnieje, wartością inicjalizacji jest INF. Zatem spełniony jest również warunek (2).

Indukcyjny dowód poprawności algorytmu Dikstry

48

□ Krok indukcyjny:

- Założmy, że warunki (1) i (2) są spełnione po ustaleniu k wierzchołków oraz niech v będzie (k+1) ustalonym wierzchołkiem.



Hipotetyczna krótsza droga do v
wiodąca przez w i u.

Indukcyjny dowód poprawności algorytmu Dikstry

49

□ Krok indukcyjny:

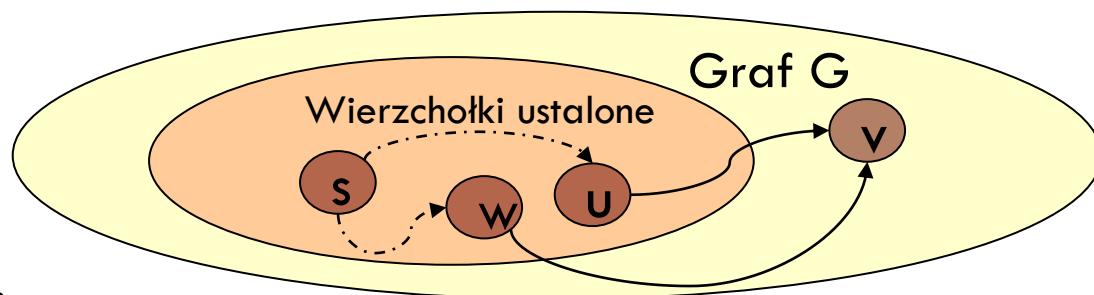
- Warunek (1) jest wciąż spełniony ponieważ $\text{dist}(v)$ jest najmniejszą długością drogi z s do v.
 - Założmy, że tak nie jest. Musiała by więc istnieć hipotetyczna krótsza droga do v wiodąca przez w i u. Jednakże wierzchołek v został obrany jako $k+1$ ustalony, co oznacza, że w tym momencie $\text{dist}(u)$ nie może być mniejsze od $\text{dist}(v)$, gdyż wówczas jako $(k+1)$ wierzchołek wybrany zostałby wierzchołek u.
 - Na podstawie warunku (2) hipotezy indukcyjnej wiadomo, że $\text{dist}(u)$ jest minimalna długością drogi specjalnej wiodącej do u. Jednak droga z s przez w do u jest drogą specjalną, tak więc jej długość równa jest co najmniej $\text{dist}(u)$. Stąd domniemana krótsza droga z s do v wiodąca przez w i u ma długość równą co najmniej $\text{dist}(v)$, ponieważ pierwsza jej część, - z s do u – ma długość $\text{dist}(u)$, a $\text{dist}(u) \geq \text{dist}(v)$. Stąd warunek (1) jest spełniony dla $k+1$ wierzchołków.

Indukcyjny dowód poprawności algorytmu Dikstry

50

□ Krok indukcyjny:

- Warunek (1) jest wciąż spełniony ponieważ $\text{dist}(v)$ jest najmniejszą długością drogi z s do v .



Dwie możliwości określenia
przedostatniego wierzchołka
w drodze specjalnej do u .

Indukcyjny dowód poprawności algorytmu Dikstry

51

□ Krok indukcyjny (cd):

- Teraz należy pokazać, że warunek (2) jest spełniony po dodaniu do wierzchołków ustalonych wierzchołka v .
 - Weźmy pod uwagę pewien wierzchołek u , który wciąż pozostaje nieustalony po dodaniu v do wierzchołków ustalonych. W najkrótszej drodze specjalnej do u musi istnieć pewien wierzchołek przedostatni. Wierzchołkiem tym może być zarówno v , jak i pewien inny wierzchołek w .
 - Przyjmijmy, że wierzchołkiem przedostatnim jest v . Długość drogi z s przez v do u wynosi $\text{dist}(v) + \text{wartość etykiety } v \rightarrow u$.
 - Przyjmijmy, że wierzchołkiem przedostatnim jest w . Na podstawie warunku (1) hipotezy indukcyjnej można stwierdzić, że najkrótsza droga z s do w składa się jedynie z wierzchołków, które zostały ustalone przed v , stąd wierzchołek v nie występuje w tej drodze.
 - A więc długość drogi specjalnej do u się nie zmienia po dodaniu v do wierzchołków ustalonych.
 - Ponieważ w momencie ustalania wierzchołka v przeprowadzona jest operacja dostosowywania $\text{dist}(u)$, warunek (2) jest spełniony.

Algorytmy znajdowania najkrótszych dróg

52

- Jeśli potrzebne jest poznanie minimalnych odległości między wszystkimi parami wierzchołków w grafie o n wierzchołkach, które posiadają etykiety o wartościach nieujemnych, można uruchomić algorytm Dijkstry dla każdego z n wierzchołków jako wierzchołka źródłowego.
- Czas wykonania algorytmu Dijkstry wynosi $O(m \log n)$, gdzie m oznacza większą wartość z liczby wierzchołków i liczby krawędzi. Znalezienie w ten sposób minimalnych odległości między wszystkimi parami wierzchołków zajmuje czas rzędu $O(m n \log n)$.
- Jeśli m jest bliskie swojej maksymalnej wartości $m \approx n^2$ to można skorzystać z implementacji algorytmu Dijkstry który działa w czasie $O(n^2)$. Wykonanie go n razy daje czas rzędu $O(n^3)$.

Algorytmy znajdowania najkrótszych dróg

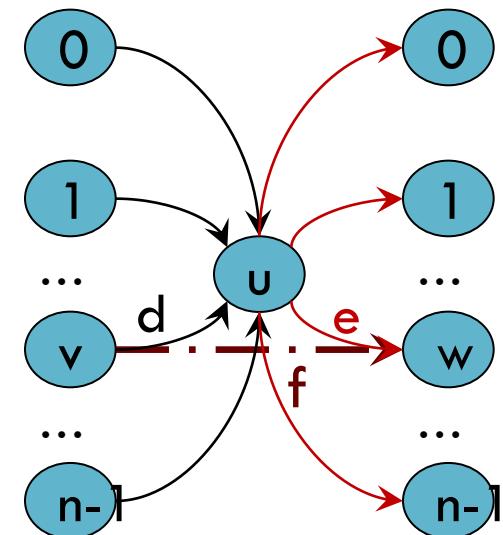
53

- Istnieje inny algorytm znajdowania minimalnych odległości między wszystkimi parami wierzchołków, noszący nazwę algorytmu Floyda-Warshalla.
- Jego wykonanie zajmuje czas rzędu $O(n^3)$. Operuje na macierzach sąsiedztwa a nie listach sąsiedztwa i jest koncepcyjnie prostszy.

Algorytm Floyda-Warshalla

54

- Podstawa algorytmu jest działanie polegające na rozpatrywaniu po kolejni każdego wierzchołka grafu jako elementu centralnego (ang. pivot).
- Kiedy wierzchołek **u** jest elementem centralnym, staramy się wykorzystać fakt, że **u** jest wierzchołkiem pośrednim między wszystkimi parami wierzchołków.
- Dla każdej pary wierzchołków, na przykład **v** i **w**, jeśli suma etykiet krawędzi **(v, u)** oraz **(u, w)** (na rysunku **d+e**) , jest mniejsza od bieżąco rozpatrywanej etykiety **f** krawędzi wiodącej od **v** do **w**, to wartość **f** jest zastępowana wartością **d+e**.

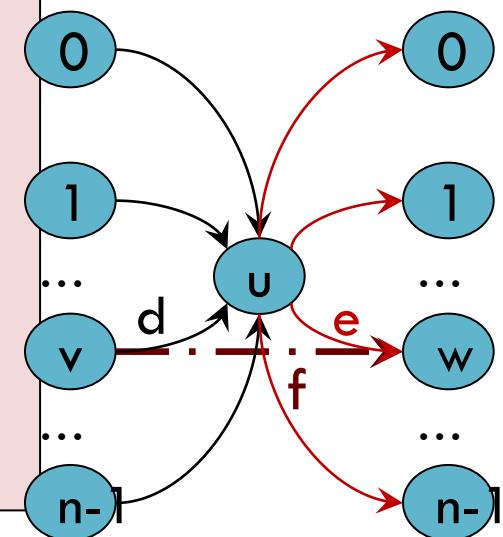


Algorytm Floyda-Warshalla

55

Node u, v, w;

```
for (v = 0; w < MAX; v++)
    for (w=0; w < MAX; w++)
        dist[v][w] = edge[v][w];
for (u=0; u< MAX; u++)
    for (v=0; v< MAX; v++)
        for (w=0; w<MAX; w++)
            if( dist[v][u]+dist[u][w] < dist[v][w])
                dist[v][w] = dist [v][u] + dist [u][w];
```

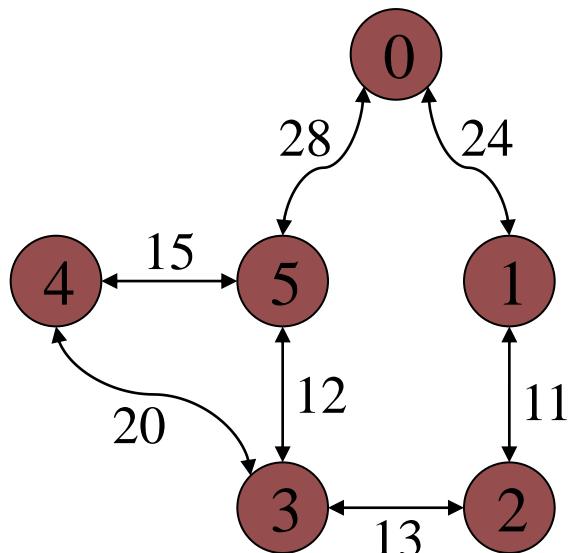


edge[v][w] –etykieta krawędzi, wierzchołki numerowane

Algorytm Floyda-Warshalla

56

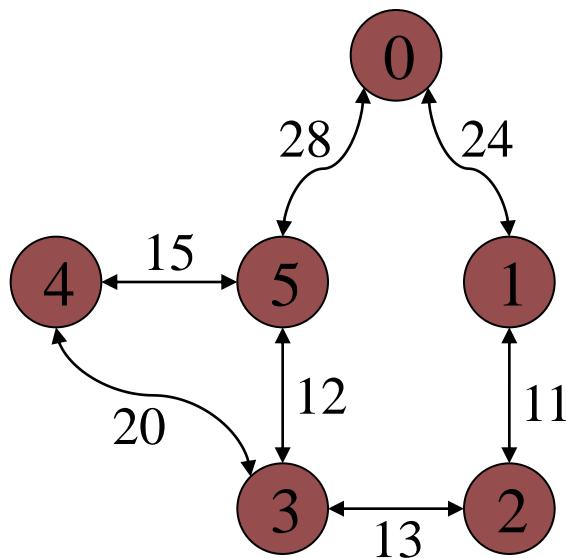
Macierz która odzwierciedla początkową postać macierzy odległości (ang. *dist*)



	0	1	2	3	4	5
0	0	24	INF	INF	INF	28
1	24	0	11	INF	INF	INF
2	INF	11	0	13	INF	INF
3	INF	INF	13	0	20	12
4	INF	INF	INF	20	0	15
5	28	INF	INF	12	15	0

Algorytm Floyda-Warshalla

57

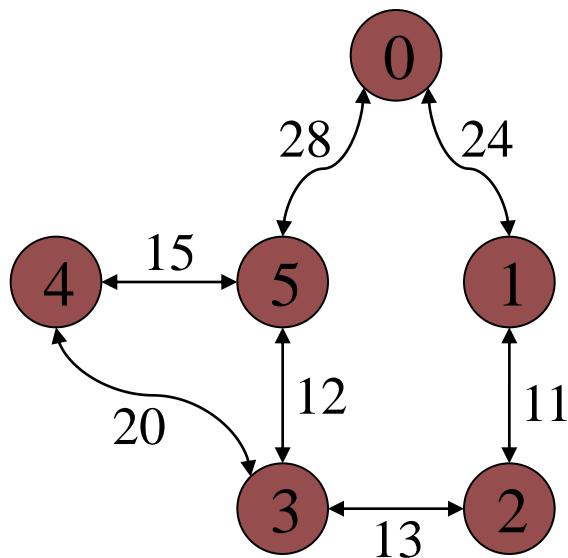


Macierz odległości, po użyciu
wierzchołka **0** jako elementu
centralnego

	0	1	2	3	4	5
0	0	24	INF	INF	INF	28
1	24	0	11	INF	INF	52
2	INF	11	0	13	INF	INF
3	INF	INF	13	0	20	12
4	INF	INF	INF	20	0	15
5	28	52	INF	12	15	0

Algorytm Floyda-Warshalla

58



Macierz odległości, po użyciu
wierzchołka **1** jako elementu
centralnego

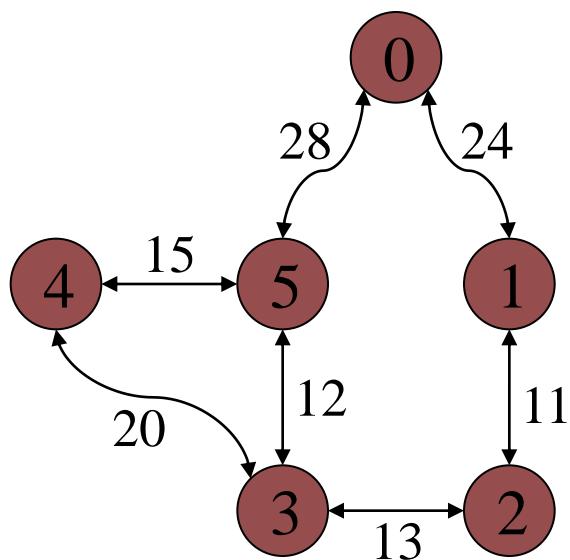
	0	1	2	3	4	5
0	0	24	35	INF	INF	28
1	24	0	11	INF	INF	52
2	35	11	0	13	INF	63
3	INF	INF	13	0	20	12
4	INF	INF	INF	20	0	15
5	28	52	63	12	15	0

itd... itd...

Algorytm Floyda-Warshalla

59

Końcowa postać macierzy
odległości



	0	1	2	3	4	5
0	0	24	35	40	43	28
1	24	0	11	24	44	52
2	35	11	0	13	33	25
3	40	24	13	0	20	12
4	43	44	33	20	0	15
5	28	36	25	12	15	0

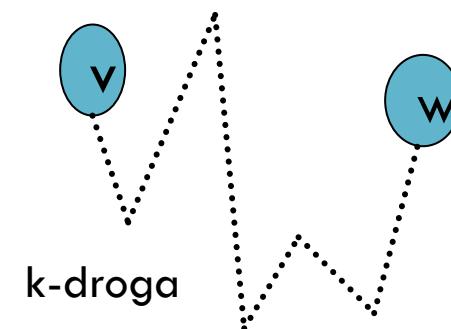
Uzasadnienie poprawności algorytmu Floyda-Warshalla

60

- **Na dowolnym etapie działania algorytmu Floyda-Warshalla odległość z wierzchołka v do wierzchołka w stanowi długość najkrótszej z tych dróg, które wiodą jedynie przez wierzchołki użyte dotąd jako elementy centralne.**
- **Ponieważ wszystkie wierzchołki zostają w końcu użyte jako elementy centralne, elementy $\text{dist}[v][w]$ zawierają po zakończeniu działań minimalne długości wszystkich możliwych dróg.**

numery wyższe od k

numery niższe od k



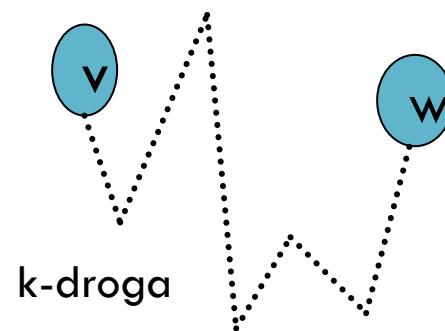
Uzasadnienie poprawności algorytmu Floyda-Warshalla

61

- Definiujemy **k-drogę z wierzchołka v do wierzchołka w jako drogę z v do w taką, że żaden jej wierzchołek pośredni nie ma numeru wyższego od k.**
- Należy zauważyć, że nie ma ograniczenia odnośnie tego, że v lub w mają mieć wartość k lub mniejszą.
- $k=-1$ oznacza że droga nie posiada wierzchołków pośrednich.

numery wyższe od k

numery niższe od k



Dowód indukcyjny

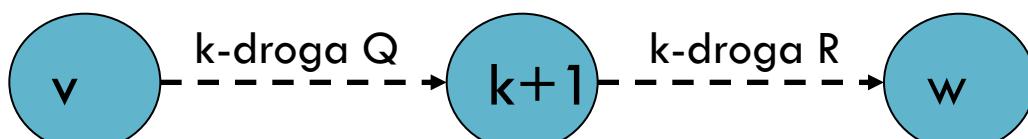
62

□ Teza indukcyjna $S(k)$:

- jeżeli etykiety krawędzi mają wartości nieujemne, to po przebiegu k – pętli, element $\text{dist}[v][w]$ ma wartość najkrótszej k – drogi z v do w lub ma wartość INF, jeżeli taką drogę nie istnieje.

□ Podstawa:

- Podstawą jest warunek $k = -1$. Krawędzie i drogi składające się z pojedynczego wierzchołka są jedynymi (-1) drogami.



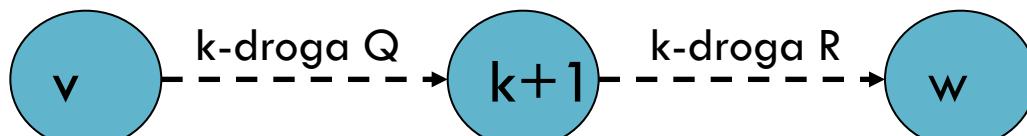
k-drogę P można rozbić na dwie k-drogi, Q oraz R.

Dowód indukcyjny

63

□ Krok indukcyjny:

- Założmy ze $S(k)$ jest spełnione i rozważmy co się dzieje z elementami $\text{dist}[v][w]$ w czasie $k+1$ przebiegu pętli.
- Założmy, że P jest najkrótszą $(k+1)$ – drogą wiodącą z v do w . Mamy do czynienia z dwoma przypadkami, w zależności czy droga P prowadzi przez wierzchołek $k+1$.



k-drogę P można rozbić na dwie k-drogi, Q oraz R .

Dowód indukcyjny

64

□ Przypadek 1:

- Jeżeli P jest k -drogą, to znaczy, kiedy P nie wiedzie przez wierzchołek $k+1$, to na podstawie hipotezy indukcyjnej wartość elementu $\text{dist}[v][w]$ jest równa długości P po zakończeniu k -tej iteracji. Nie można zmienić wartości $\text{dist}[v][w]$ podczas przebiegu wykonywanego dla wierzchołka $k+1$ traktowanego jako element centralny, gdyż nie istnieją żadne krótsze $(k+1)$ -drogi.

□ Przypadek 2:

- Jeżeli P jest $(k+1)$ -droga, można założyć, że P przechodzi przez wierzchołek $k+1$ tylko raz, gdyż cykl nigdy nie może spowodować zmniejszenia odległości (przy założeniu że wszystkie etykiety mają wartości nieujemne).
- Stąd droga P składa się z k -drogi Q , wiodącej od wierzchołka v do $k+1$, oraz k -drogi R , wiodącej od wierzchołka $k+1$ do w . Na podstawie hipotezy indukcyjnej wartości elementów $\text{dist}[v][k+1]$ oraz $\text{dist}[k+1][w]$ będą długosćiami dróg odpowiednio, Q i R , po zakończeniu k -tej iteracji.

Dowód indukcyjny

65

- **Ostatecznie wnioskujemy, że w $(k+1)$ przebiegu, wartością elementu $\text{dist}[v][w]$ staje się długość najkrótszej $(k+1)$ -drogi dla wszystkich wierzchołków v oraz w .**
- **Jest to twierdzenie $S(k+1)$, co oznacza koniec kroku indukcyjnego.**
- **Weźmy teraz, że $k=n-1$. Oznacza to, że wiemy iż po zakończeniu wszystkich n przebiegów, wartość $\text{dist}[v][w]$ będzie minimalną odlegością dowolnej $(n-1)$ -drogi wiodącej z wierzchołka v do w . Ponieważ każda droga jest $(n-1)$ drogą, więc $\text{dist}[v][w]$ jest minimalną długością drogi wiodącej z wierzchołka v do w .**

Podsumowanie

66

PROBLEM	ALGORYTM(Y)	CZAS WYKONANIA
Minimalne drzewo rozpinające	Algorytm Kruskala	$O(m \log n)$
Znajdowanie cykli	Przeszukiwanie w głąb	$O(m)$
Uporządkowanie topologiczne	Przeszukiwanie w głąb	$O(m)$
Osiągalność w przypadku pojedynczego źródła	Przeszukiwanie w głąb	$O(m)$
Spójne składowe	Przeszukiwanie w głąb	$O(m)$
Najkrótsza droga dla pojedyncz. źródła	Algorytm Dijkstry	$O(m \log n)$
Najkrótsza droga dla wszystkich par	Algorytm Dijkstry Algorytm Floyda	$O(m n \log n)$ $O(n^3)$

Pytania do egzaminu

67

- 1) **Co to jest grafowy model danych? Omów podstawową terminologię dotyczącą grafów**
- 2) **Co to znaczy implementacja przy pomocy macierzy sąsiedztwa? Zilustruj przykładem.**
- 3) **Co to znaczy implementacja przy pomocy listy sąsiedztwa? Zilustruj przykładem.**
- 4) **Co to jest składowa spójna grafu? Co to jest drzewo rozpinające?**
- 5) **Na czym polega algorytm Kruskala dla znajdowania minimalnego drzewa rozpinającego?**
- 6) **Na czym polega sortowanie topologiczne grafu?**
- 7) **Na czym polega algorytm Dikstry, jaka jest jego złożoność obliczeniowa?**
- 8) **Na czym polega algorytm Floyda, jak jest jego złożoność obliczeniowa?**

TEORETYCZNE PODSTAWY INFORMATYKI

Wykład 6 – część I

2

Modele
danych:
zbiory

- **Podstawowe definicje**
- **Operacje na zbiorach**
- **Prawa algebraiczne**
- **Struktury danych**
 - ▣ **Lista jednokierunkowa**
 - ▣ **Wektor własny**
 - ▣ **Tablica mieszająca**

Zbiór

3

- **Zbiór jest najbardziej podstawowym modelem danych w matematyce. Wszystkie pojęcia matematyczne, od drzew po liczby rzeczywiste można wyrazić za pomocą specjalnego rodzaju zbioru.**
- **Jest więc naturalne że jest on również podstawowym modelem danych w informatyce.**
- **Dotychczas wykorzystaliśmy to pojęcie mówiąc o słowniku, który także jest rodzajem zbioru na którym możemy wykonywać tylko określone operacje: wstawiania, usuwania i wyszukiwania.**

Podstawowe definicje

4

- W matematyce pojęcie **zbioru** nie jest zdefiniowane wprost.
- Zamiast tego, podobnie jak punkt czy prosta w geometrii, **zbiór jest zdefiniowany za pomocą swoich własności**.
- W szczególności istnieje pojęcie **przynależności**, które jest sensowne tylko i wyłącznie dla zbiorów. Jeśli **S** jest **zbiorem** oraz **x** jest **czymkolwiek**, zawsze możemy odpowiedzieć na pytanie
„**Czy x należy do zbioru S?**”
- Zbiór **S** składa się więc z wszystkich takich elementów **x**, dla których **x należy do zbioru S**.

Podstawowe definicje

5

□ Notacja:

- Wyrażenie $x \in S$ oznacza, że element x należy do zbioru S .
- Jeśli elementy x_1, x_2, \dots, x_n należą do zbioru S i żadne inne, to możemy zapisać:
$$S = \{x_1, x_2, \dots, x_n\}$$
- Każdy x musi być inny, nie możemy umieścić w zbiorze żadnego elementu dwa lub więcej razy. Kolejność ułożenia elementów w zbiorze jest jednak całkowicie dowolna.
- Zbiór pusty, oznaczamy symbolem \emptyset , jest zbiorem do którego nie należą żadne elementy.
 - Oznacza to że $x \in \emptyset$ jest zawsze fałszywe.

Podstawowe definicje

6

- **Definicja za pomocą abstrakcji:**
 - Wyliczenie elementów należących do zbioru nie jest jedynym sposobem jego definiowania. Bardzo wygodne jest wyjście od definicji że istnieje zbiór S oraz że jego elementy spełniają własność P , tzn. $\{x : x \in S \text{ oraz } P(x)\}$ czyli „zbiór takich elementów x należących do zbioru S , które spełniają własność P .
- **Równość zbiorów:**
 - Dwa zbiory są równe (czyli są tym samym zbiorem), jeśli zawierają te same elementy.
- **Zbiory nieskończone:**
 - Zwykle wygodne jest przyjęcie założenia że zbiory są skończone. Czyli że istnieje pewna skończona liczba N taka, że nasz zbiór zawiera dokładnie N elementów. Istnieją jednak również zbiory nieskończone np. liczb naturalnych, całkowitych, rzeczywistych, itd.

Operacje na zbiorach

7

Operacje często wykonywane na zbiorach:

- **Suma:** dwóch zbiorów S i T , zapisywana $S \cup T$, czyli zbiór zawierający elementy należące do zbioru S lub do zbioru T .
- **Przecięcie (iloczyn):** dwóch zbiorów S i T , zapisywana $S \cap T$, czyli zbiór zawierający należące elementy do zbioru S i do zbioru T .
- **Różnica:** dwóch zbiorów S i T , zapisywana $S \setminus T$, czyli zbiór zawierający tylko te elementy należące do zbioru S , które nie należą do zbioru T .

Operacje na zbiorach

8

- Jeżeli S i T są zdarzeniami w przestrzeni probabilistycznej,
 - suma, przecięcie i różnica mają naturalne znaczenie,
 - $S \cup T$ jest zdarzeniem polegającym na zajęciu zdarzenia S lub T ,
 - $S \cap T$ jest zdarzeniem polegającym na zajęciu zdarzenia S i T ,
 - $S \setminus T$ jest zdarzeniem polegającym na zajęciu zdarzenia S ale nie T ,
 - Jeśli S jest zbiorem obejmującym całą przestrzeń probabilistyczną, $S \setminus T$ jest dopełnieniem zbioru T .

Prawa algebraiczne

9

- **Prawo przemienności i łączności dla sumy zbiorów** określają, że możemy obliczyć sumę wielu zbiorów, wybierając je w dowolnej kolejności.
- **Wynikiem zawsze będzie taki sam zbiór elementów, czyli takich które należą do jednego lub więcej zbiorów będących operandami sumy.**

Prawo przemienności dla sumy:

$$(S \cup T) = (T \cup S)$$

Prawo łączności dla sumy:

$$(S \cup (T \cup R)) = ((S \cup T) \cup R)$$

Prawo przemienności dla przecięcia:

$$(S \cap T) = (T \cap S)$$

Prawo łączności dla przecięcia:

$$(S \cap (T \cap R)) = ((S \cap T) \cap R)$$

Prawa algebraiczne

10

- Przecięcie dowolnej liczby zbiorów nie zależy od kolejności ich grupowania

Przykład: Prawo rozdzielności przecięcia względem sumy:

$$(S \cap (T \cup R)) = ((S \cap T) \cup (S \cap R))$$

Przykład: Prawo rozdzielności sumy względem przecięcia:

$$(S \cup (T \cap R)) = ((S \cup T) \cap (S \cup R))$$

Prawa algebraiczne

11

Prawo łączności dla sumy i różnicy:

$$(S \setminus (T \cup R)) = ((S \setminus T) \cup (S \setminus R))$$

Prawo rozdzielności różnicy względem sumy:

$$((S \cup T) \setminus R) = ((S \setminus R) \cup (T \setminus R))$$

- Zbiór pusty jest elementem neutralnym sumy:
 $(S \cup \emptyset) \equiv S$
- Idempotencja sumy: $(S \cup S) = S$
- Idempotencja przecięcia: $(S \cap S) = S$
- $(S \setminus S) \equiv \emptyset$
- $(\emptyset \setminus S) \equiv \emptyset$
- $(\emptyset \cap S) \equiv \emptyset$

Prawa algebraiczne

12

■ Relacja podzbioru:

- Istnieje relacja zawierania się jednego zbioru w drugim zbiorze, co oznacza że wszystkie elementy pierwszego są również elementami drugiego.

■ Zbiór potęgowy:

- $P(S)$ zbioru S to zbiór wszystkich podzbiorów zbioru S .
- Jeśli $S = \{1,2,3\}$ to:
- $P(S) = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}\}.$

Obiekty niepodzielne

13

- To nie jest pojęcie z teorii zbiorów ale bardzo wygodne dla dyskusji o strukturach danych i algorytmach opartych na zbiorach.
- Zakładamy istnienie pewnych **obiektów niepodzielnych**, które nie są zbiorami. Obiektem niepodzielnym może być element zbioru, jednak nic nie może należeć do samego obiektu niepodzielnego. Podobnie jak zbiór pusty, obiekt niepodzielny nie może zawierać żadnych elementów. Zbiór pusty jest jednak zbiorem, obiekt niepodzielny nim nie jest.
- Kiedy mówimy o strukturach danych, często wygodne jest wykorzystanie **skomplikowanych typów** danych jako typów **obiektów niepodzielnych**. Obiektami niepodzielnymi mogą więc być struktury lub tablice, które przecież wcale nie mają „niepodzielnego” charakteru.

Implementacja zbioru na strukturze listy

14

- **Suma, przecięcie i różnica:**
 - Podstawowe operacje na zbiorach, np. suma, mogą wykorzystywać jako przetwarzaną strukturę danych listę jednokierunkową, chociaż właściwa technika przetwarzania zbiorów powinna się nieco różnić od stosowanej przez nas dla list.
 - W szczególności posortowanie elementów wykorzystywanych list znaczco skraca czas wykonywania operacji sumy, przecięcia i różnicy zbiorów.

Zbiory a listy

15

- Istotne różnice między pojęciami:
 - zbiór $S = \{x_1, x_2, \dots, x_n\}$
 - lista $L = \{x_1, x_2, \dots, x_n\}$:
- Kolejność elementów w zbiorze jest nieistotna (a dla listy jest istotna).
- Elementy należące do zbioru nie mogą się powtarzać (a dla listy mogą).

Zbiory a listy

16

Zbiory jako nieposortowane listy:

- Wyznaczenie sumy, przecięcia czy różnicy zbiorów o rozmiarach m i n wymaga czasu $O(m n)$.
 - Aby stworzyć listę U reprezentującą np. sumę pary S i T , musimy rozpocząć od skopiowania listy reprezentującej zbiór S do początkowo pustej listy U .
 - Następnie każdy element listy ze zbioru T musimy sprawdzić aby przekonać się, czy nie znajduje się on na liście U .
 - Jeśli nie to dodajemy ten element do listy U .

Zbiory a listy

17

Zbiory jako posortowane listy

- Operacje wykonujemy znacznie szybciej jeżeli elementy są posortowane. Za każdym razem porównujemy ze sobą tylko dwa elementy (po jednym z każdej listy).
 - Wyznaczenie sumy, przecięcia czy różnicy zbiorów o rozmiarach m i n wymaga czasu $O(m+n)$.
 - Jeżeli listy nie były pierwotnie posortowane to sortowanie list zajmuje $O(m \log m + n \log n)$.
 - Operacja ta może nie być szybsza niż $O(m n)$ jeśli ilość elementów list jest bardzo różna.
 - Jeżeli liczby m i n są porównywalne to $O(m \log m + n \log n) < O(m n)$

Implementacja zbiorów oparta na wektorze własnym

18

- Definiujemy **uniwersalny zbiór U** w którym zawierają się wszystkie zbiory na których będziemy przeprowadzać operacje. Np. talia kart (zbiór 52 kart) jest uniwersalny dla różnych możliwych zbiorów kart.
- Porządkujemy elementy zbioru **U** w taki sposób, by każdy element tego zbioru można było związać z **unikatową „pozycją”**, będącą liczbą całkowitą od **0 do n-1** (gdzie n jest liczbą elementów w zbiorze uniwersalnym). Liczba elementów w zbiorze **S** jest m.
- Wówczas, zbiór **S** zawierający się w zbiorze **U**, możemy reprezentować za pomocą **wektora własnego** złożonego z zer i jedynek – dla każdego elementu **x** należącego do zbioru **U**, jeśli **x** należy także do zbioru **S**, odpowiadającą temu elementowi pozycja zawiera wartość **1**; jeśli **x** nie należy do **S**, na odpowiedniej pozycji mamy wartość **0**.

Implementacja zbiorów oparta na wektorze własnym

19

- Czas potrzebny na wykonanie operacji sumy, przecięcia i różnicy jest $O(n)$.
- Jeśli przetwarzane zbiory są dużą częścią zbioru uniwersalnego to jest to dużo lepsze niż $O(m \log m)$ (posortowanie listy) lub $O(m^2)$ (nieposortowane listy).
- Jeśli $m \ll n$ to jest to oczywiście nieefektywne.
- Ta implementacja również niepraktyczna jeżeli wymaga zbyt dużego U .

Przykład z kartami

20

□ Przykład:

- Niech U będzie talią kart.
- Umawiamy się że porządkujemy karty w talii w następujący sposób:
 - Kolorami: trefl, karo, kier, pik.
 - W każdym kolorze wg schematu: as, 2, 3, ..., walet, dama, król.
- Przykładowo pozycja:
 - as trefl to 0,
 - król trefl to 12,
 - as karo to 13,
 - walet pik to 49.

Przykład z kartami

21

Przykład z jabłkami

22

	Odmiana	Kolor	Dojrzewa
0	Delicious	czerwony	późno
1	Granny Smith	zielony	wcześnie
2	Jonathan	czerwony	wcześnie
3	McIntosh	czerwony	wcześnie
4	Gravenstein	czerwony	późno
5	Pippin	zielony	późno

$$\text{Czerwone} = 101110$$

$$\text{Wcześnie} = 011100$$

$$\text{Czerwone} \cup \text{Wcześnie} = 111110$$

$$\text{Czerwone} \cap \text{Wcześnie} = 001100$$

Czas potrzebny do wyznaczenia sumy, przecięcia i różnicy jest proporcjonalny do długości wektora własnego

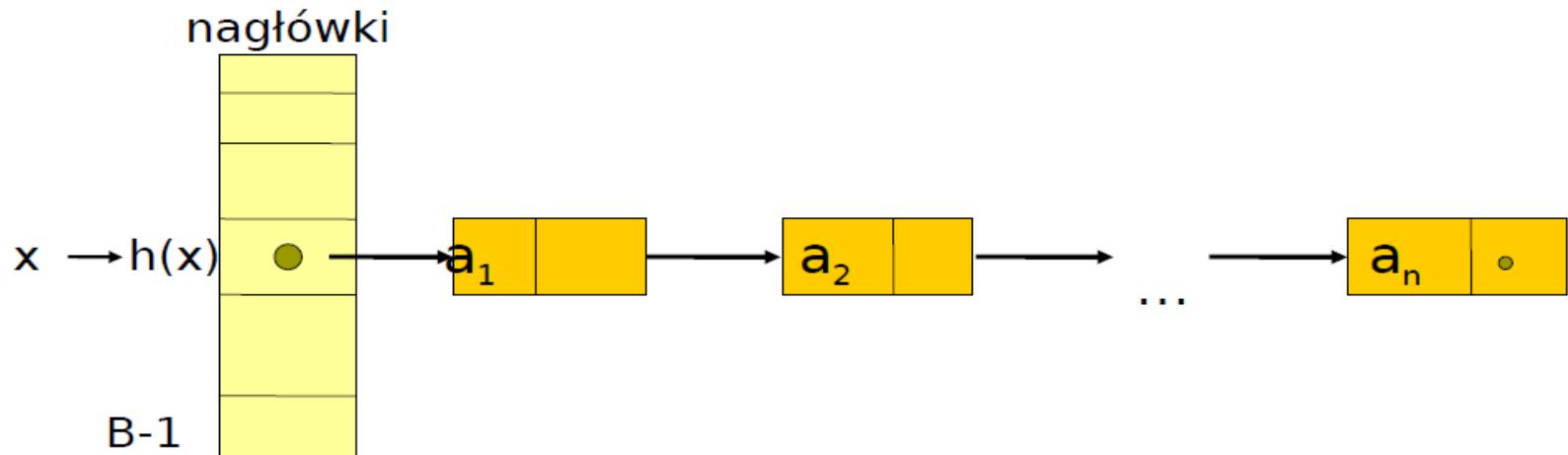
Struktura danych oparta na tablicy mieszającej

23

- Reprezentacja słownika oparta o wektor własny, jeśli tylko możliwa, umożliwiłaby bezpośredni dostęp do miejsca w którym element jest reprezentowany.
- Nie możemy jednak wykorzystywać zbyt dużych zbiorów uniwersalnych ze względu na pamięć i czas inicjalizacji.
- Np. słownik dla słów złożonych z co najwyżej 10 liter.
Ile możliwych kombinacji: $26^{10} + 26^9 + \dots + 26 = 10^{14}$ możliwych słów.
Faktyczny słownik: to tylko około 10^6 .
Co robimy?
- Grupujemy, każda grupa to jedna komórka z „nagłówkiem” + lista jednokierunkowa z elementami należącymi do grupy.
- Taką strukturę nazywamy tablicą mieszającą (ang. hash table)

Struktura danych tablicy mieszającej

24



- Istnieje **funkcja mieszająca** (ang. **hash function**), która jako argument pobiera element x i zwraca liczbę całkowitą z przedziału 0 do $B-1$, gdzie B jest liczbą komórek w tablicy mieszającej.
- Wartością zwracaną przez $h(x)$ jest komórka, w której umieszczały element x .
- Ważne aby funkcja $h(x)$ „mieszala”, tzn. aby komórki zawierały tę samą przybliżoną liczbę elementów.

Struktura danych tablicy mieszającej

25

- Każda komórka składa się z listy jednokierunkowej, w której przechowujemy wszystkie elementy zbioru wysłanego do tej komórki przez funkcje mieszające.
- Aby odnaleźć element x obliczamy wartość $h(x)$, która wskazuje na numer komórki.
- Jeśli tablica mieszająca zawiera element x , to możemy go znaleźć przeszukując listę która znajduje się w tej komórce.
- Tablica mieszająca pozwala na wykorzystanie reprezentacji zbiorów opartej na liście (wolne przeszukiwanie), ale dzięki podzieleniu zbioru na B komórek, czas przeszukiwania jest $\sim 1/B$ potrzebnego do przeszukiwania całego zbioru.
- W szczególności może być nawet $O(1)$, czyli taki jak w reprezentacji zbioru opartej na wektorze własnym.

Implementacja słownika oparta na tablicy mieszającej

26

- **Aby wstawić, usunąć lub wyszukać element x w słowniku zaimplementowanym przy użyciu tablicy mieszającej, musimy zrealizować proces złożony z trzech kroków.**
 - **wyznaczyć właściwą komórkę przy użyciu funkcji $h(x)$**
 - **wykorzystać tablice wskaźników do nagłówków w celu znalezienia listy elementów znajdującej się w komórce wskazanej przez $h(x)$**
 - **wykonać na tej liście operacje tak jakby reprezentowała cały zbiór**

Podsumowanie

27

- **Pojęcie zbioru ma zasadnicze znaczenie w informatyce.**
- **Najczęściej wykonywanymi operacjami na zbiorach są: suma, przecięcie oraz różnica.**
- **Do modyfikowania i upraszczania wyrażeń złożonych ze zbiorów i zdefiniowanych na nich operacji możemy wykorzystywać prawa algebraiczne.**

Podsumowanie

28

- **Listy jednokierunkowe, wektory własne oraz tablice mieszające to trzy najprostsze sposoby reprezentowania zbiorów w języku programowania.**
- **Listy jednokierunkowe oferują największą elastyczność w przypadku większości operacji na zbiorach, nie zawsze są jednak rozwiązaniem najbardziej efektywnym.**
- **Wektory własne są najszybszym rozwiązaniem dla pewnych operacji, mogą jednak być wykorzystywane tylko w sytuacjach, gdy zbiór uniwersalny jest mały.**
- **Często złotym środkiem są tablice mieszające, które zapewniają zarówno oszczędne wykorzystanie pamięci jak i satysfakcjonujący czas wykonania operacji.**

Wykład 6 – część II

29

Modele danych: drzewa

- **Podstawowa terminologia**
- **Rekurencyjna definicja**
- **Drzewa zaetykietowane**
 - **Drzewa wyrażeń**
- **Struktura danych dla drzew**
 - **Reprezentacje drzewa**
 - **Rekurencja w drzewach**
- **Drzewa binarne**
- **Drzewa przeszukiwania binarnego**
- **Drzewa binarne częściowo uporządkowane**
- **Kolejka priorytetowa**
- **Zrównoważone drzewa częściowo uporządkowane i kopce**

Model danych oparty na drzewach

30

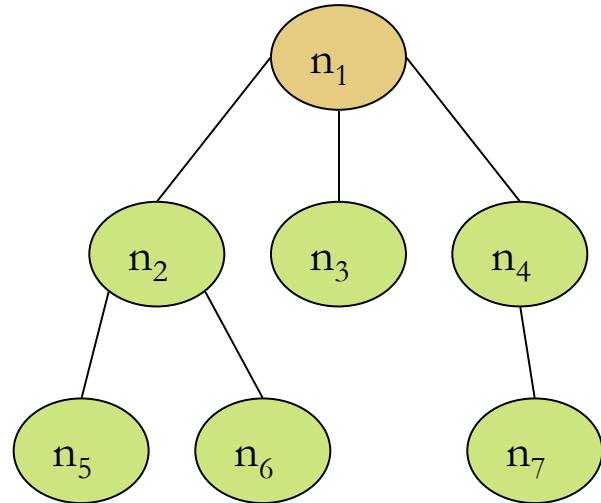
- Istnieje wiele sytuacji w których przetwarzane informacje mają strukturę hierarchiczną lub zagnieżdżoną, jak drzewo genealogiczne lub diagram struktury organizacyjnej.

- Abstrakcje modelujące strukturę hierarchiczną nazywamy drzewem – jest to jeden z najbardziej podstawowych modeli danych w informatyce.

Podstawowa terminologia

31

- Drzewa są zbiorami punktów, zwanych węzłami lub wierzchołkami, oraz połączeń, zwanych krawędziami.
- Krawędź łączy dwa różne węzły.



n_1 = rodzic n_2, n_3, n_4

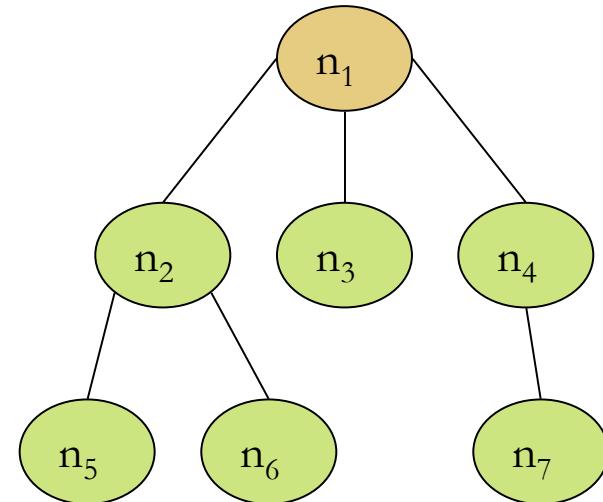
n_2 = rodzic n_5, n_6

n_6 = dziecko n_2

Podstawowa terminologia

32

- Aby struktura zbudowana z węzłów połączonych krawędziami była drzewem musi spełniać pewne warunki:
 - W każdym drzewie wyróżniamy jeden węzeł zwany korzeniem n_1 (ang. root)
 - Każdy węzeł c nie będący korzeniem jest połączony krawędzią z innym węzłem zwany rodzicem p (ang. parent) węzła c. Węzeł c nazywamy także dzieckiem (ang. child) węzła p.
 - Każdy węzeł c nie będący korzeniem ma dokładnie jednego rodzica.
 - Każdy węzeł ma dowolną liczbę dzieci.
 - Drzewo jest spójne (ang. connected) w tym sensie że jeżeli rozpoczęliśmy analizę od dowolnego węzła c nie będącego korzeniem i przejdziemy do rodzica tego węzła, następnie do rodzica tego rodzica, itd., osiągniemy w końcu korzeń.



n_1 = rodzic n_2 , n_3 , n_4

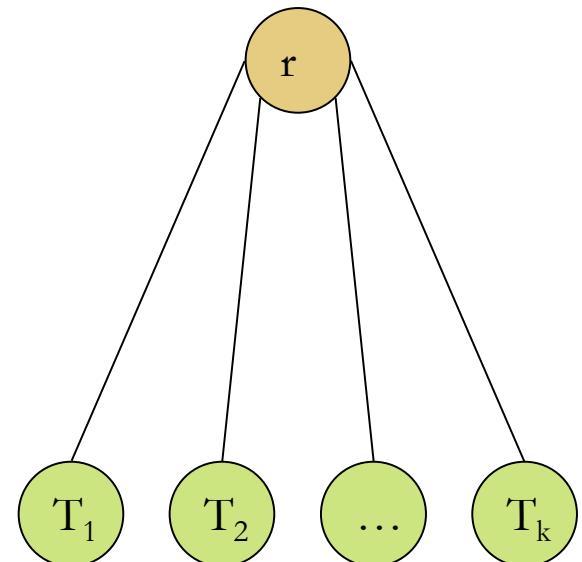
n_2 = rodzic n_5 , n_6

n_6 = dziecko n_2

Rekurencyjna definicja drzew

33

- **Podstawa:** Pojedynczy węzeł n jest drzewem.
Mówimy że n jest korzeniem drzewa złożonego z jednego węzła.
- **Indukcja:** Niech r będzie nowym węzłem oraz niech T_1, T_2, \dots, T_k będą drzewami zawierającymi odpowiednio korzenie c_1, c_2, \dots, c_k . Założmy że żaden węzeł nie występuje więcej niż raz w drzewach T_1, T_2, \dots, T_k , oraz że r , będący „nowym” węzłem, nie występuje w żadnym z tych drzew.
Nowe drzewo T tworzymy z węzła r i drzew T_1, T_2, \dots, T_k w następujący sposób:
 - węzeł r staje się korzeniem drzewa T ;
 - dodajemy krawędzi, po jednej łączącej r z każdym z węzłów c_1, c_2, \dots, c_k , otrzymując w ten sposób strukturę w której każdy z tych węzłów jest dzieckiem korzenia r . Inny sposób interpretacji tego kroku to uczynienie z węzła r rodzica każdego z korzeni drzew T_1, T_2, \dots, T_k .



Podstawowa terminologia

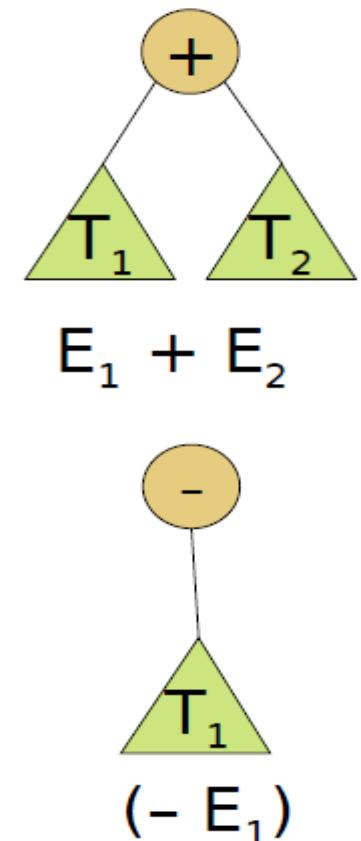
34

- Relacje rodzic-dziecko można w naturalny sposób rozszerzyć do relacji **przodków i potomków**.
- **Ścieżką** nazywamy ciąg węzłów, takich że poprzedni jest rodzicem następnego. Węzły na ścieżce to potomkowie (przodkowie). Jeżeli ciąg węzłów (n_1, n_2, \dots, n_k) jest ścieżką, to **długość ścieżki** wynosi **k-1** (długość ścieżki dla pojedynczego węzła wynosi 0). Jeżeli ścieżka ma **długość ≥ 1** , to węzeł m_1 nazywamy właściwym przodkiem węzła m_k , a węzeł m_k właściwym potomkiem węzła m_1 .
- W dowolnym drzewie **T**, dowolny węzeł **n** wraz z jego potomkami nazywamy **poddzewem**.
- **Liściem** (ang. leaf) nazywamy węzeł drzewa który nie ma potomków.
- **Węzeł wewnętrzny** to taki węzeł który ma jednego lub większą liczbę potomków.
- **Wysokość drzewa** to długość najdłuższej ścieżki od korzenia do liścia.
- **Głębokość węzła** to długość drogi od korzenia do tego węzła.

Drzewa zaetykietowane i drzewa wyrażeń

35

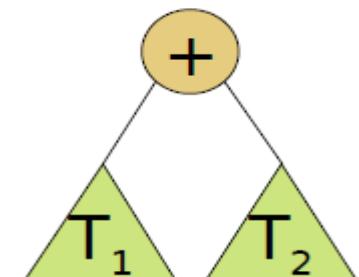
- Drzewo zaetykietowane to takie w którym z każdym węzłem drzewa związana jest jakaś etykieta lub wartość. Możemy reprezentować wyrażenia matematyczne za pomocą drzew zaetykietowanych.



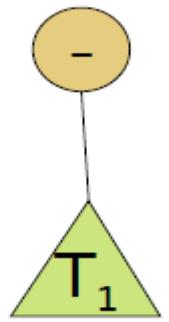
Drzewa zaetykietowane i drzewa wyrażeń

36

- Definicja drzewa zaetykietowanego dla wyrażeń arytmetycznych zawierających operandy dwuargumentowe $+, -, \cdot, /$ oraz operator jednoargumentowy $-$.
- Podstawa: Pojedynczy operand niepodzielny jest wyrażeniem. Reprezentujące go drzewo składa się z pojedynczego węzła, którego etykietą jest ten operand.
- Indukcja: Jeśli E_1 oraz E_2 są wyrażeniami reprezentowanymi odpowiednio przez drzewa T_1 , T_2 , wyrażenie $(E_1 + E_2)$ reprezentowane jest przez drzewo którego korzeniem jest węzeł o etykiecie $+$. Korzeń ten ma dwoje dzieci, którego korzeniami są odpowiednio korzenie drzew T_1 , T_2 .



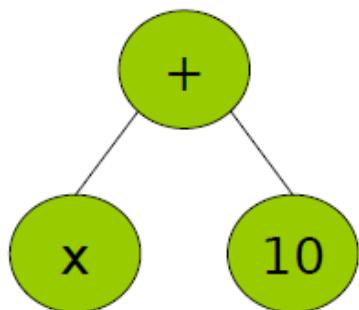
$$E_1 + E_2$$



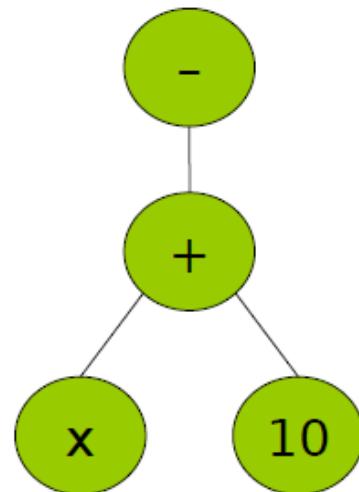
$$(- E_1)$$

Konstrukcja drzew wyrażeń

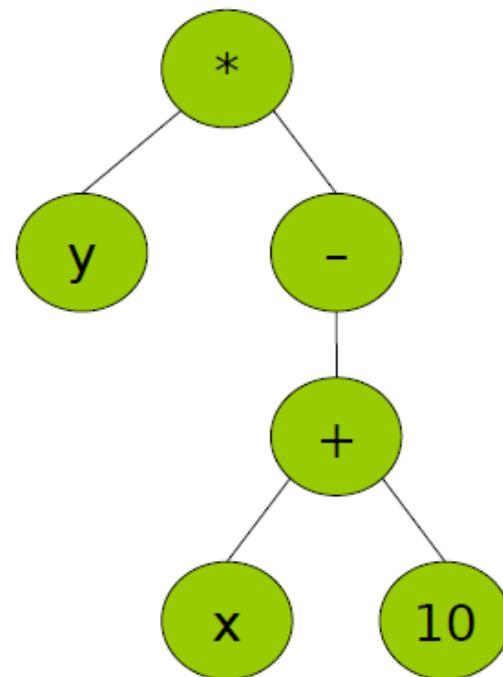
37



$(x + 10)$



$(- (x + 10))$



$(y * - (x + 10))$

Struktura danych dla drzew

38

- Do reprezentowania drzew możemy używać wiele różnych **struktur danych**. Wybór odpowiedniej struktury zależy od konkretnych operacji które planujemy wykonać na budowanych drzewach.
- **Przykład:**
 - Jeżeli jedynym planowanym działaniem jest lokalizowanie rodziców danych węzłów, zupełnie wystarczającą będzie struktura składająca się z etykiety węzła i wskaźnika do struktury reprezentującej jego rodzica.
- W ogólności, węzły drzewa możemy reprezentować za pomocą struktur, których pola łączą węzły w drzewa w sposób podobny do łączenia węzła za pomocą wskaźnika do struktury korzenia.

Struktura danych dla drzew

39

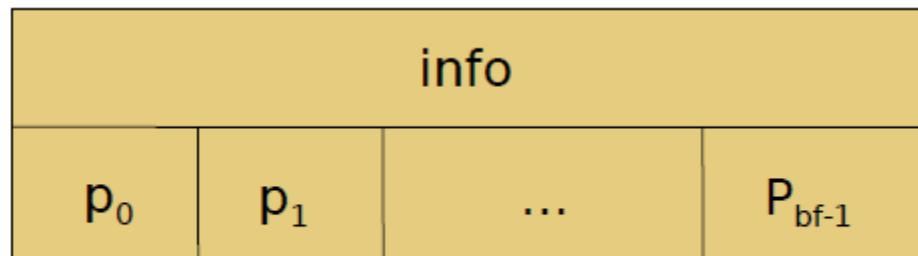
- Kiedy mówimy o reprezentowaniu drzew, w pierwszej kolejności mamy na myśli sposób **reprezentowania węzłów**.
- Różnica między reprezentacjami dotyczy miejsca w pamięci komputera gdzie przechowywana jest **struktura zawierająca węzły**.
 - W języku C możemy stworzyć przestrzeń dla struktur reprezentujących wierzchołki za pomocą funkcji malloc ze standartowej biblioteki stdlib.h, co powoduje, że do umieszczonych w pamięci węzłów mamy dostęp tylko za pomocą wskaźników.
 - Rozwiązaniem alternatywnym jest stworzenie **tablicy struktur** i wykorzystanie jej elementów do reprezentowania węzłów. Możemy uzyskać dostęp do węzłów nie wykorzystując ścieżek w drzewie. Wadą jest z góry określony rozmiar tablicy (musi istnieć ograniczenie maksymalnego rozmiaru drzewa).

Tablica wskaźników jako reprezentacja drzewa

40

- Jednym z najprostszych sposobów reprezentowania drzewa jest wykorzystanie dla każdego węzła struktury składającej się z pola lub pól reprezentujących etykietę oraz tablicy wskaźników do dzieci tego węzła.
- Info reprezentuje etykietę węzła.
- Stała bf jest rozmiarem tablicy wskaźników. Reprezentuje maksymalną liczbę dzieci dowolnego węzła, czyli czynnik rozgałęzienia (ang. branching factor).
- i-ty element tablicy reprezentującej węzeł zawiera wskaźnik do i-tego dziecka tego węzła.
- Brakujące połączenia możemy reprezentować za pomocą wskaźnika pustego **NULL**.

```
typedef struct NODE *pNODE  
struct NODE{  
    int info;  
    pNODE children[BF];  
};
```



Reprezentacja drzewa

41

- Wykorzystujemy **listę jednokierunkową** reprezentującą dzieci węzła. Przestrzeń zajmowana przez listę jest dla węzła proporcjonalna do liczby jego dzieci.
- Znaczącą wadą tego rozwiązania jest efektywność czasowa – uzyskanie dostępu do **i-tego** dziecka wymaga czasu **$O(i)$** , ponieważ musimy przejść przez całą listę o długości **$i-1$** , by dostać się do **i-tego** węzła.
- Dla porównania, jeżeli zastosujemy tablicę wskaźników do dzieci, do **i-tego** dziecka dostajemy się w czasie **$O(1)$** , niezależnie od wartości **i**.

Reprezentacje drzewa

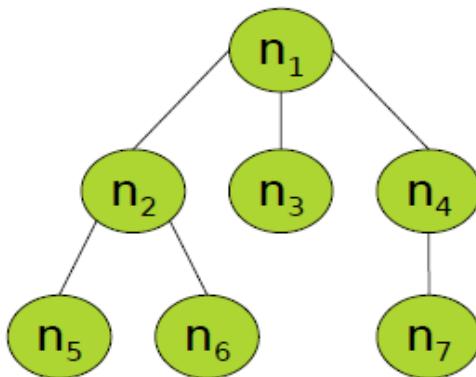
42

- W reprezentacji drzew zwanej skrajnie lewy potomek-prawy element siostrzany (ang. left-most-child-right-sibling), w każdym węźle umieszczamy jedynie wskaźniki do skrajnie lewego dziecka; węzeł nie zawiera wskaźników do żadnego ze swoich pozostałych dzieci.
- Aby odnaleźć drugi i wszystkie kolejne dzieci węzła n, tworzymy listę jednokierunkową tych dzieci w której każde dziecko c wskazuje na znajdujące się bezpośrednio po jego prawej stronie dziecko węzła n.
- Wskazany węzeł nazywamy prawym elementem siostrzanym węzła c.

Reprezentacje drzewa

43

Drzewo złożone z 7 węzłów



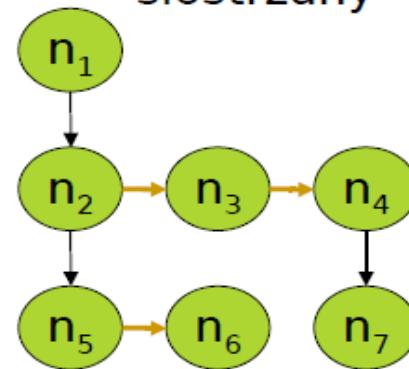
info - etykieta

leftmostChild - informacja o węźle

rightSibling - część listy

jednokierunkowej dzieci rodzica tego węzła

Reprezentacja skrajnie lewy potomek-prawy element siostrzany



```
typedef struct NODE *pNODE;
struct NODE{
    int info;
    pNODE leftmostChild, rightSibling;
};
```

Reprezentacje drzewa

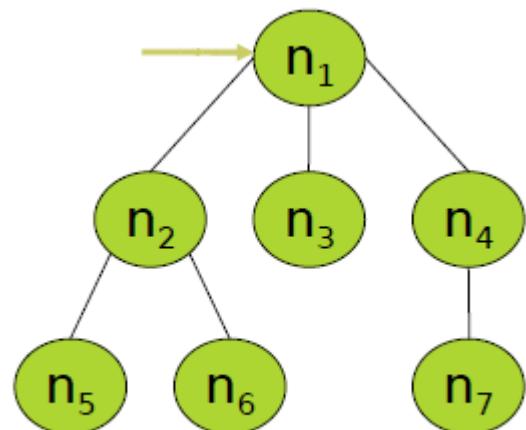
44

- **Reprezentacja oparta na tablicy wskaźników** umożliwia nam dostęp do i-tego dziecka dowolnego węzła w czasie $O(1)$. Taka reprezentacja wiąże się jednak ze znacznym marnotrawstwem przestrzeni pamięciowej, jeśli tylko kilka węzłów ma wiele dzieci. W takim wypadku większość wskaźników w tablicy `children` będzie równa `NULL`.
- **Reprezentacja skrajnie lewy potomek-prawy element siostrzany wymaga mniejszej przestrzeni pamięciowej.** Nie wymaga również istnienia maksymalnego czynnika rozgałęzienie węzłów. Możemy reprezentować węzły z dowolna wartością tego czynnika, nie modyfikując jednocześnie struktury danych.

Rekurencja w drzewach

45

- Użyteczność drzew wynika z liczby możliwych **operacji rekurencyjnych**, które możemy na nich wykonać w naturalny i jasny sposób (chcemy drzewa przeglądać).
- Prosta rekurencja zwraca etykiety węzłów **w porządku wzwyżnym** (ang. **pre-order listing**), czyli: korzeń, lewe poddrzewo, prawe poddrzewo.
- Inną powszechnie stosowaną metodą do przeglądania węzłów drzewa jest tzw. **przeszukiwanie wsteczne** (ang. **post-order listing**), czyli lewe poddrzewo, prawe poddrzewo, korzeń.



Drzewa binarne

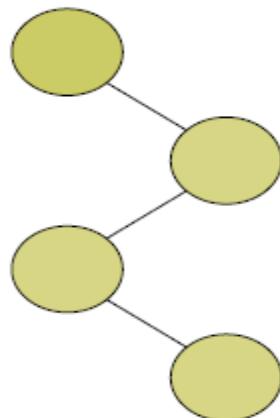
46

- W drzewie binarnym węzeł może mieć co najwyżej dwoje bezpośrednich potomków.
- Rekurencyjna definicja drzewa binarnego:
 - Podstawa:
 - Drzewo puste jest drzewem binarnym.
 - Indukcja:
 - Jeśli r jest węzłem oraz T_1, T_2 są drzewami binarnymi, istnieje drzewo binarne z korzeniem r, lewym poddrzewem T_1 i prawym poddrzewem T_2 . Korzeń drzewa T_1 jest lewym dzieckiem węzła r, chyba że T_1 jest drzewem pustym. Podobnie korzeń drzewa T_2 jest prawym dzieckiem węzła r, chyba że T_2 jest drzewem pustym.
 - Większość terminologii wprowadzonej przy okazji drzew stosuje się oczywiście też do drzew binarnych.
- Różnica: drzewa binarne wymagają rozróżnienia lewego od prawego dziecka, zwykle drzewa tego nie wymagają. Drzewa binarne to NIE są zwykłe drzewa, w których węzły mogą mieć co najwyżej dwójkę dzieci.

Drzewa binarne

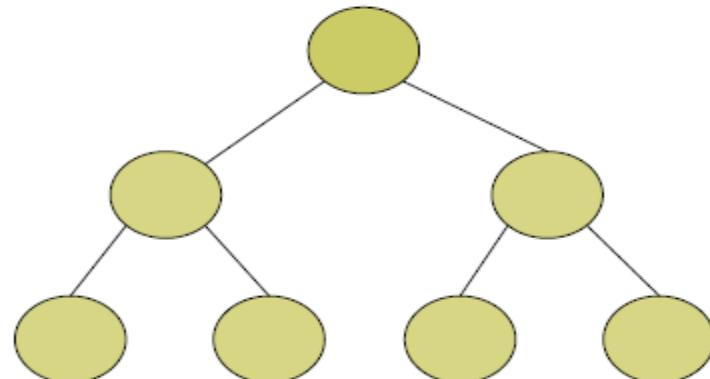
47

Zdegenerowane drzewo binarne



Wysokość drzewa złożonego z k -węzłów to $k-1$.
Czyli $h = O(k)$.
Operacje insert, delete, find wymagają średnio $O(k)$.

Pełne drzewo binarne



Drzewo o wysokości h ma $k=2^{h+1}-1$ węzłów.
Czyli $h = O(\log k)$.
Operacje insert, delete, find wymagają średnio $O(\log k)$.

Drzewa przeszukiwania binarnego

48

- **Jest to zaetykietowane drzewo binarne dla którego etykiety należą do zbioru w którym możliwe jest zdefiniowanie relacji mniejszości.**
- **Dla każdego węzła x spełnione są następujące własności:**
 - **wszystkie węzły w lewym poddrzewie mają etykiety mniejsze od etykiety węzła x**
 - **wszystkie w prawym poddrzewie mają etykiety większe od etykiety węzła x .**

Drzewa przeszukiwania binarnego

49

□ Wyszukiwanie elementu:

□ Podstawa:

- Jeśli drzewo **T** jest puste, to na pewno nie zawiera elementu **x**.
- Jeśli **T** nie jest puste i szukana wartość **x** znajduje się w korzeniu, drzewo zawiera **x**.

□ Indukcja:

- Jeśli **T** nie jest puste, ale nie zawiera szukanego elementu **x** w korzeniu, niech **y** będzie elementem w korzeniu drzewa **T**.
- Jeśli **x < y**, szukamy wartości **x** tylko w lewym poddrzewie korzenia **y**.
- Jeśli **x > y**, szukamy wartości **x** tylko w prawym poddrzewie korzenia **y**.

Własność drzewa przeszukiwania binarnego gwarantuje, że szukanej wartości **x na pewno nie ma w poddrzewie, którego nie przeszukujemy**

Drzewa przeszukiwania binarnego

50

□ Wstawianie elementu:

□ Podstawa:

- Jeśli drzewo T jest drzewem pustym, zastępujemy T drzewem składającym się z pojedynczego węzła zawierającego element x.
- Jeśli drzewo T nie jest puste oraz jego korzeń zawiera element x, to x znajduje się już w drzewie i nie wykonujemy żadnych dodatkowych kroków.

□ Indukcja:

- Jeśli T nie jest puste i nie zawiera elementu x w swoim korzeniu, niech y będzie elementem w korzeniu drzewa T.
- Jeśli $x < y$, wstawiamy wartość x do lewego poddrzewa T.
- Jeśli $x > y$, wstawiamy wartość x do prawego poddrzewa T.

Drzewa przeszukiwania binarnego

51

Usuwanie elementu:

- Usuwanie elementu x z drzewa przeszukiwania binarnego jest zadaniem nieco bardziej skomplikowanym od znajdowania czy wstawiania danego elementu. Musimy zachować własność drzewa przeszukiwania binarnego.
- Lokalizujemy x , oznaczmy węzeł w którym się on znajduje poprzez v .
 - Jeśli drzewo nie zawiera x to nie robimy nic.
 - Jeżeli v jest liściem to go usuwamy.
 - Jeśli v jest wewnętrznym węzłem i węzeł ten ma tylko jedno dziecko, przypisujemy węzłowi v wartość dziecka v , a następnie usuwamy dziecko v . (W ten sposób że dziecko dziecka v , staje się dzieckiem v , a rodzicem dziecka dziecka v staje się v).
 - Jeżeli węzeł v ma dwoje dzieci, oznaczmy poprzez y najmniejszą wartość w prawym poddrzewie v . Następnie przypisujemy węzłowi v wartość y , i usuwamy y z prawego poddrzewa v .

Słownik

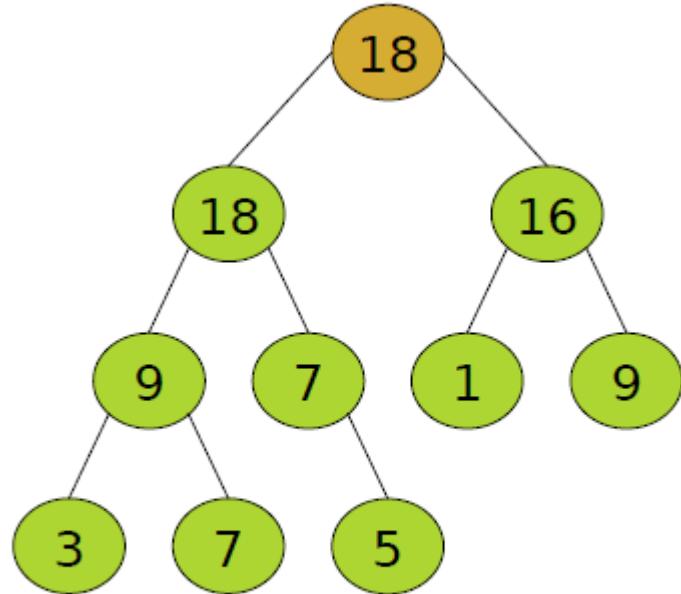
52

- Często stosowaną w programach komputerowych modelem danych jest zbiór, na którym chcemy wykonywać operacje:
 - wstawianie nowych elementów do zbioru (ang. **insert**)
 - usuwanie elementów ze zbioru (ang. **delete**)
 - wyszukiwanie jakiegoś elementu w celu sprawdzenia, czy znajduje się w danym zbiorze (ang. **find**)
- Taki zbiór będziemy nazywać **słownikiem** (niezależnie od tego jakie elementy zawiera). Drzewo przeszukiwania binarnego umożliwia stosunkowo efektywną implementację słownika.
- Czas wykonania każdej z operacji na słowniku reprezentowanym przez drzewo przeszukiwania binarnego złożone z **n węzłów** jest proporcjonalny do **wysokości tego drzewa h**.

Drzewa binarne częściowo uporządkowane

53

- Jest to zaetykietowane drzewo binarne o następujących właściwościach:
 - Etykietami węzłów są elementy z przypisanymi priorytetami; priorytet może być wartością elementu lub przynajmniej jednego z jego komponentów.
 - Element przechowywany w węźle musi mieć co najmniej taki duży priorytet jak element znajdujący się w dzieciach tego węzła. Element znajdujący się w korzeniu dowolnego poddrzewa jest więc największym elementem tego poddrzewa.



Kolejka priorytetowa

54

- Inny typ danych to **zbiór elementów**, z których każdy jest związany z określonym **priorytetem**. Przykładowo, elementy mogą być strukturami, zaś priorytet może być wartością jednego z pól takiej struktury. Chcemy wykonywać operacje:
 - **wstawianie nowych elementów do zbioru (ang. insert)**
 - **znalezienie i usuniecie ze zbioru elementu o najwyższym priorytecie (ang. deletemax)**
- Taki zbiór będziemy nazywać **kolejką priorytetową** (niezależnie od tego jakie elementy zawiera).
- **Drzewo binarne częściowo uporządkowane** umożliwia stosunkowo efektywną implementację kolejki priorytetowej.
- **Efektywna (używając kopca) tzn. $O(\log n)$.**

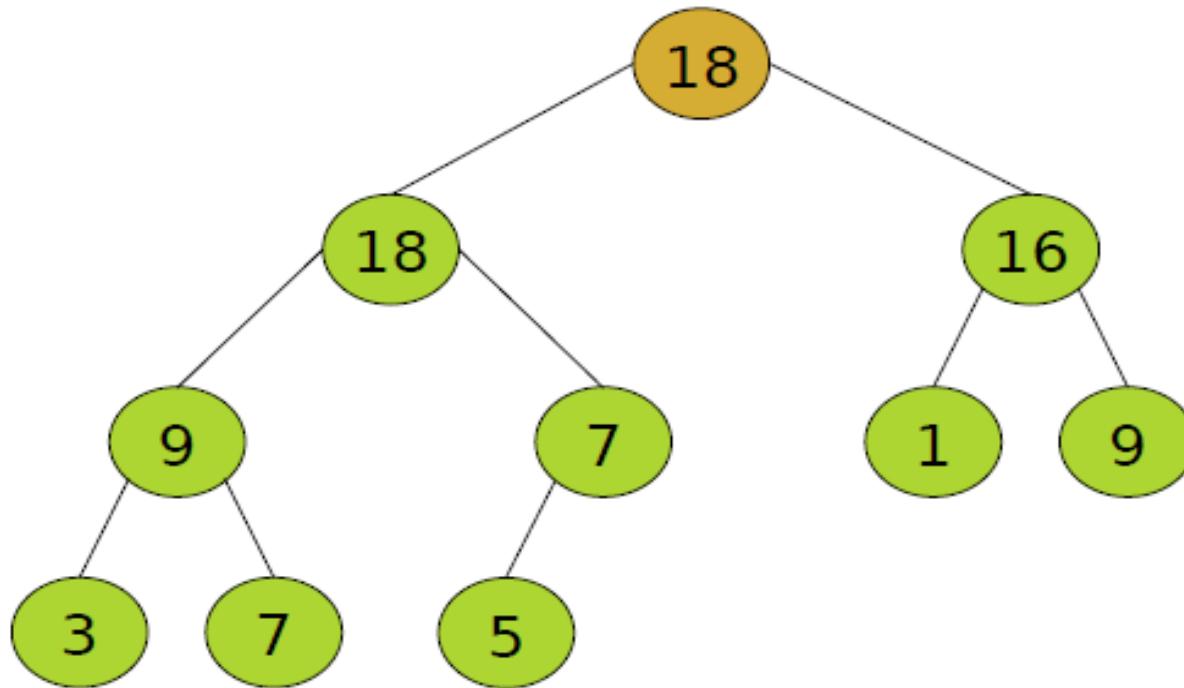
Zrównoważone drzewa częściowo uporządkowane i kopce

55

- Mówimy że drzewo uporządkowane jest **zrównoważone** (ang. **balanced**), jeśli na wszystkich poziomach poza najniższym zawiera wszystkie możliwe węzły oraz liście na najniższym poziomie są ułożone od lewej strony. Spełnienie tego warunku oznacza, że jeśli drzewo składa się z n węzłów, to żadna ścieżka od korzenia do któregokolwiek z tych węzłów nie jest dłuższa niż $\log_2 n$.
- **Zrównoważone drzewa częściowo uporządkowane** można implementować za pomocą tablicowej struktury danych zwanej **kopcem** (ang. **heap**), która umożliwia szybką i zwięzłą implementację kolejek priorytetowych. Kopiec jest to po prostu **tablica A**, której sposób indeksowania reprezentujemy w specyficzny sposób. Zapisuje się kolejne poziomy, zawsze porządkując od lewej do prawej.

Zrównoważone drzewa częściowo uporządkowane i kopce

56

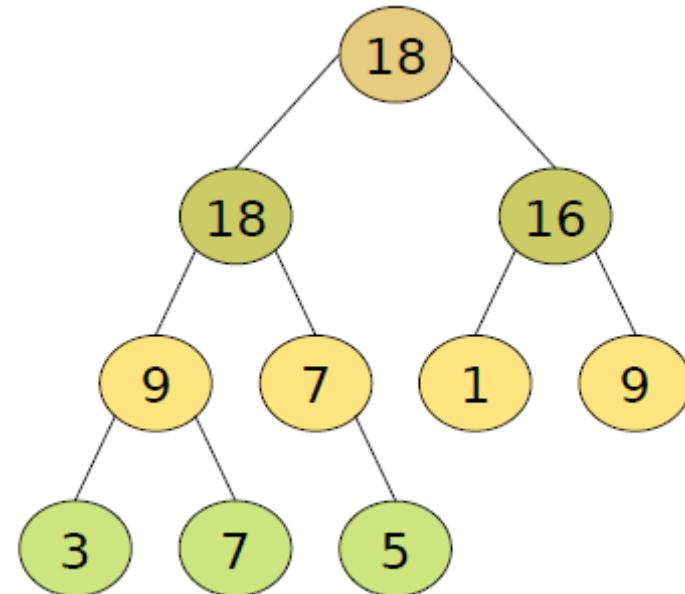


Kopiec dla zrównoważonego częściowo uporządkowanego drzewa.

57

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]	A[10]
18	18	16	9	7	1	9	3	7	5

- Rozpoczynamy od korzenia A[1]; nie wykorzystujemy A[0].
- Po korzeniu zapisujemy kolejne poziomy, w każdym poziomie węzły porządkujemy od lewej do prawej.
- Lewe dziecko korzenia znajduje się w A[2]; prawe dziecko korzenia umieszczamy w A[3].
- W ogólności, lewe dziecko węzła zapisane w A[i] znajduje się w A[2i], prawe dziecko tego samego węzła znajduje się w A[2i+1], jeśli oczywiście te dzieci istnieją w drzewie uporządkowanym.
- Taka reprezentacja jest możliwa dzięki właściwościom drzewa zrównoważonego.
- Z właściwości drzewa częściowo uporządkowanego wynika, że jeśli A[i] ma dwójkę dzieci, to A[i] jest co najmniej tak duże, jak A[2i] i A[2i+1], oraz jeśli A[i] ma jedno dziecko, to A[i] nie jest mniejsze niż A[2i].



Operacje kolejki priorytetowej na kopcu

58

Reprezentujemy kopiec za pomocą globalnej tablicy liczb całkowitych $A[1, \dots, MAX]$.

Przypuśćmy, że mamy kopiec złożony z $n-1$ elementów, który spełnia własność drzewa częściowo uporządkowanego.

□ Operacja insert:

- Dodajemy n -ty element w $A[n]$.
- Własność drzewa uporządkowanego jest nadal spełniona we wszystkich elementach tablicy, poza (być może) elementem $A[n]$ i jego rodzicem.
- Jeśli element $A[n]$ jest większy od elementu $A[n/2]$, czyli jego rodzica, musimy wymienić te elementy ze sobą (ta operacja nazywamy sortowanie bąbelkowe w górę (ang. bubbleUp)).
- Może teraz zaistnieć konflikt z własnością drzewa częściowo uporządkowanego pomiędzy elementem $A[n/2]$ i jego rodzicem. Sprawdzamy i ewentualnie wymieniamy ich pozycje.
- Itd.

□ Operacja deletemax:

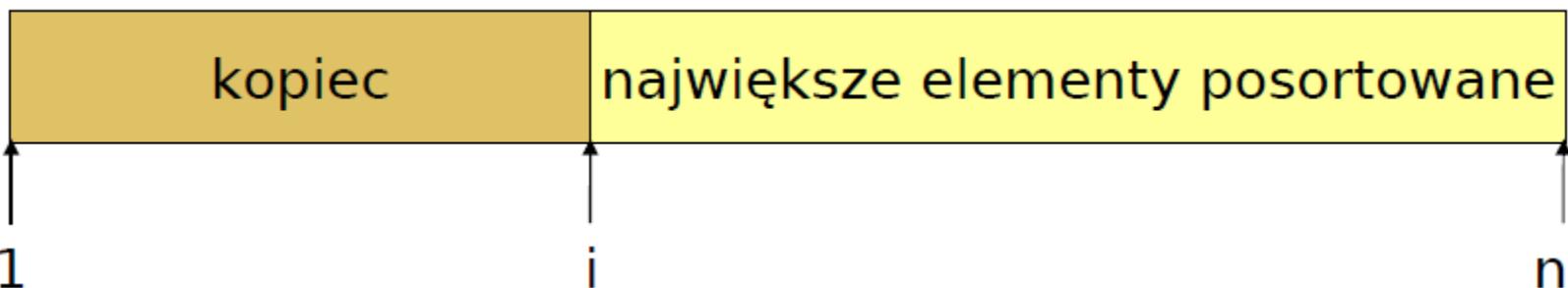
- $A[1]$ przypisujemy wartość $-\infty$, po czym wykorzystujemy analogiczną procedurę co powyżej, czyli: sortowanie bąbelkowe w dół (ang. bubbleDown) niech k oznacza pozycję w tablicy liścia do którego zejdzie wartość $-\infty$. $A[k]$ przypiszmy wartość $A[n]$. Po czym wykonajmy procedure sortowania bąbelkowego w górę. Kopiec będzie teraz reprezentowany przez tablicę $A[1, \dots, n-1]$.

Wykonywanie operacji insert i deletemax wymaga czasu $O(\log n)$.

Sortowanie przez kopcowanie

59

- Za pomocą tego algorytmu sortujemy tablice $A[1, \dots, n]$ w dwóch etapach:
 - algorytm nadaje tablicy A własność drzewa częściowo uporządkowanego
 - wielokrotnie wybiera największy z pozostałych elementów z kopca aż do momentu, w którym na kopcu znajduje się tylko jeden (najmniejszy) element co oznacza że tablica jest posortowana.



- Wykonanie operacji sortowania przez kopcowanie wymaga czasu $O(n \log n)$. Dla porównania sortowanie przez wybieranie wymaga czasu $O(n^2)$.

Poziomy implementacji

60

- Dwa abstrakcyjne typy danych:
słownik i kolejka priorytetowa
- Omówiliśmy dwie różne abstrakcyjne implementacje i wybraliśmy konkretne struktury danych dla każdej z tych abstrakcyjnych implementacji.

Abstrakcyjny typ danych	Abstrakcyjna implementacja	Struktura danych
Słownik	Drzewo przeszukiwania binarnego	Struktura lewe dziecko - prawe dziecko
Kolejka priorytetowa	Zrównoważone drzewo częściowo uporządkowane	Kopiec

Podsumowanie

61

- Ważnym modelem danych reprezentującym informacje hierarchiczne są drzewa.
- Do implementowania drzew możemy wykorzystać wiele różnych struktur danych (także takich) które wymagają połączenia tablic ze wskaźnikami. Wybór struktury danych zależy od operacji wykonywanych na drzewie.
- Dwie najważniejsze reprezentacjami węzłów drzewa są skrajnie lewy potomek-prawy element siostrzany oraz tree (tablica wskaźników do dzieci).
- Drzewa nadają się doskonale do stosowania na nich algorytmów i dowodów rekurencyjnych.
- Drzewo binarne jest jednym z wariantów modelu drzewa, w którym każdy węzeł ma (opcjonalne) lewe i prawe dziecko.
- Drzewo przeszukiwania binarnego jest zaetykietowanym drzewem binarnym, które spełnia „własność drzewa przeszukiwania binarnego”.

Podsumowanie

62

- Będący abstrakcyjnym typem danych, słownik jest zbiorem, na którym można wykonywać operacje `insert`, `delete`, `find`. Efektywna implementacja słownika to drzewo przeszukiwania binarnego.
- Innym abstrakcyjnym typem danych jest kolejka priorytetowa, czyli zbiór na którym możemy wykonywać operacje `insert` i `deletemax`. Efektywna implementacja to kopiec.
 - Drzewo częściowo uporządkowane jest zaetykietowanym drzewem binarnym spełniającym warunek, że żadna etykieta w żadnym węźle nie jest mniejsza od etykiety żadnego z jego dzieci.
 - Zrównoważone drzewo częściowo uporządkowane (węzły całkowicie wypełniają wszystkie poziomy od korzenia do najniższego, w którym zajmowane są tylko skrajnie lewe pozycje) możemy implementować za pomocą kopca. Struktura ta umożliwia implementację operacji na kolejce priorytetowej wykonywanych w czasie $O(\log n)$ oraz algorytmu sortującego, zwanego sortowaniem przez kopcowanie działającego w czasie $O(n \log n)$.

Pytania do egzaminu

63

- 1) **Jakie operacje wykonujemy na zbiorach?**
- 2) **Wytlumacz dlaczego implementacje danych typu „zbiór” przy pomocy list jednokierunkowych znaczco skraca czas wykonywania operacji „suma zbiorów”?**
- 3) **Porównaj złożoność operacji dla implementacji zbioru na liście jednokierunkowej, wektorze własnym i tablicy mieszajcej.**
- 4) **Jaki znasz struktury danych używane do reprezentacji drzew?**
- 5) **Co to jest drzewo binarne?**
- 6) **Co to jest drzewo przeszukiwania binarnego?**
- 7) **Jak są realizowane podstawowe operacje dla drzew przeszukiwania binarnego? Jakie to są operacje?**
- 8) **Czym charakteryzuje się zrównoważone drzewo częściowo uporządkowane?**
- 9) **Na czym polega sortowanie stogowe?**

TEORETYCZNE PODSTAWY INFORMATYKI

Wykład 14: Repetytorium

2

Algorytmy i
ich schematy
blokowe

- ❑ **Proste algorytmy iteracyjne**
- ❑ **Algorytmy z wykorzystaniem rekurencji**
- ❑ **Algorytmy sortujące**

Wykład na podstawie skryptu:

D. Nyk, „Algorytmy w przykładach”

http://informatyka.2ap.pl/ftp/3d/algorytmy/podrecznik_algorytmy.pdf

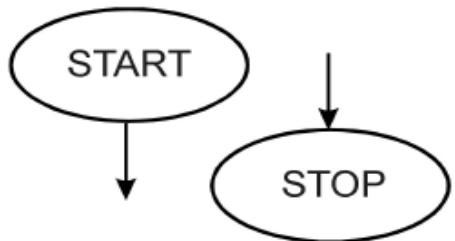
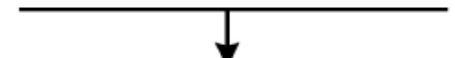
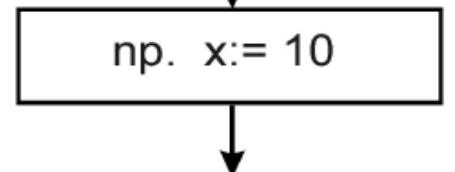
Schemat blokowy

3

- **Przedstawia algorytm w postaci symboli graficznych, podając szczegółowo wszystkie operacje arytmetyczne, logiczne, przesyłania, pomocnicze wraz z kolejnością ich wykonywania**
- **Składa się z wielu elementów z których podstawowy jest blok**
- **Poniżej przedstawione typowe podstawowe bloki programów, istnieją oczywiście jeszcze inne.**

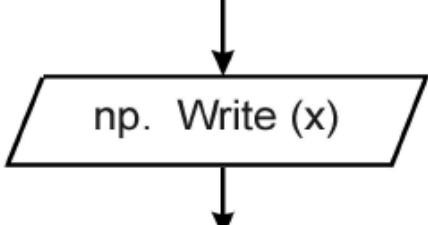
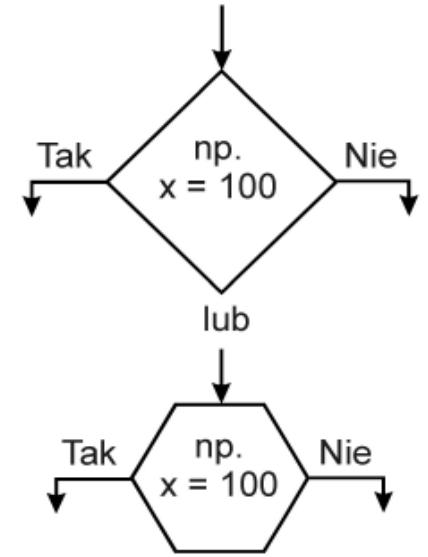
Schemat blokowy

4

Wygląd bloku	Opis
	<p>Bloki graniczne – początek i koniec algorytmu. Mają kształt owalu. Z bloku Start wychodzi tylko jedno połączenie; każdy schemat blokowy musi mieć dokładnie jeden blok START. Każdy schemat blokowy musi mieć co najmniej jeden blok STOP.</p>
	<p>Lącznik pomiędzy blokami – określa kierunek przepływu danych lub kolejność wykonywanych działań (ścieżka sterująca)</p>
	<p>Blok kolekcyjny – łączy kilka różnych dróg algorytmu</p>
 <p>np. $x := 10$</p>	<p>Blok operacyjny – zawiera operację lub grupę operacji, w których wyniku ulega zmianie wartość zmiennej (tu : nadanie zmiennej x wartości 10). Bloki operacyjne mają kształt prostokąta , wchodzi do niego jedno połączenie i wychodzi też jedno.</p>

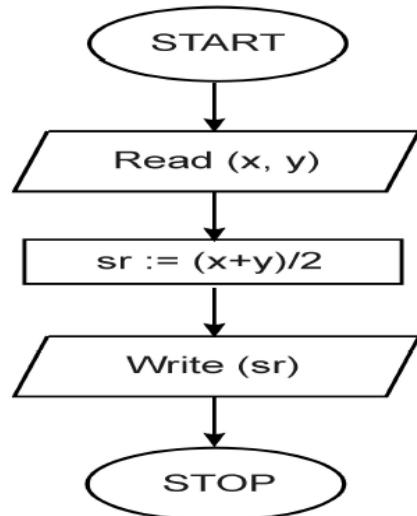
Schemat blokowy

5

	<p>Blok wejścia / wyjścia – blok odpowiedzialny za wykonanie operacji wprowadzania i wyprowadzania danych, wyników, komunikatów. Ma kształt równoległoboku, wchodzi i wychodzi z niego jedno połączenie.</p>
	<p>Blok decyzyjny – określa wybór jednej z dwóch możliwych dróg działania. Ma kształt rombu lub sześciokąta. Wchodzi do niego jedno połączenie, a wychodzą dwa : TAK – gdy warunek wpisany wewnętrz jest spełniony oraz NIE – gdy warunek wpisany wewnętrz nie jest spełniony. Wybór kształtu bloku zależy od nas.</p>

Schemat blokowy i specyfikacja programu

6



Algorytm liczenia średniej

Specyfikacją problemu algorytmicznego nazywamy dokładny opis problemu algorytmicznego, który ma zostać rozwiązyany oraz podanie informacji o danych wejściowych i wyjściowych.
Czyli przed naszym algorytmem powinien znaleźć się dodatkowy zapis :

Problem algorytmiczny : obliczenie średniej arytmetycznej dwóch liczb rzeczywistych

Dane wejściowe : $x, y \in \mathbb{R}$

Dane wyjściowe : $sr \in \mathbb{R}$ – średnia liczb x i y

Operandy i operatory

7

- Stałe i zmienne łączymy operatorami aby otrzymać wyrażenie. Stałe i zmienne nazywamy operandami.

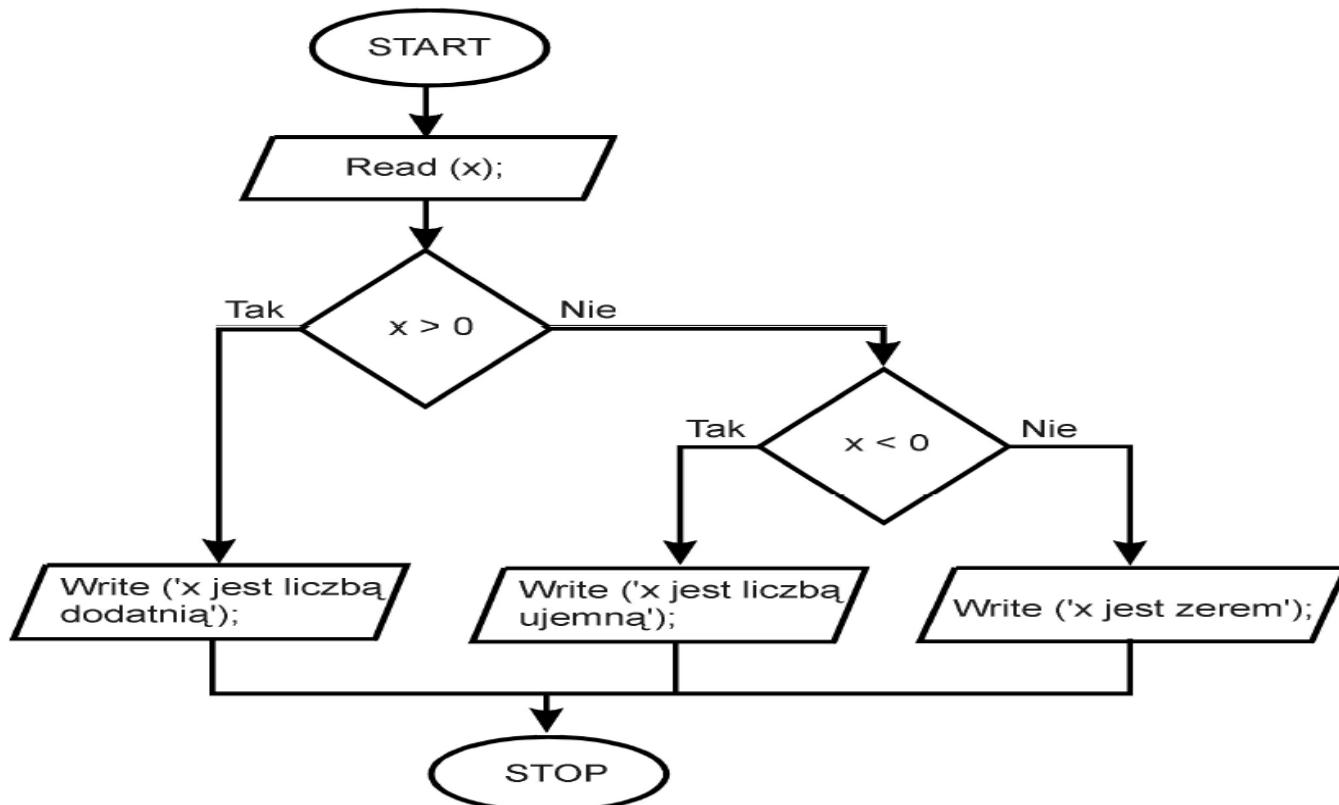
Operatory arytmetyczne		Operatory relacji	
Symbol	Znaczenie	Symbol	Znaczenie
+	dodawanie	=	równy
-	odejmowanie	>	większy
*	mnożenie	>=	większy lub równy
/	dzielenie	<	mniejszy
div	dzielenie całkowite ($3 \text{div} 2 = 1$)	<=	mniejszy lub równy
mod	reszta z dzielenia liczb całkowitych	◊	różny

Operatory logiczne

and	- koniunkcja (iloczyn zdań)	\wedge
or	- alternatywa (suma zdań)	\vee
not	- negacja (zaprzeczenie zdania)	\sim

Algorytmy z rozgałęzieniem

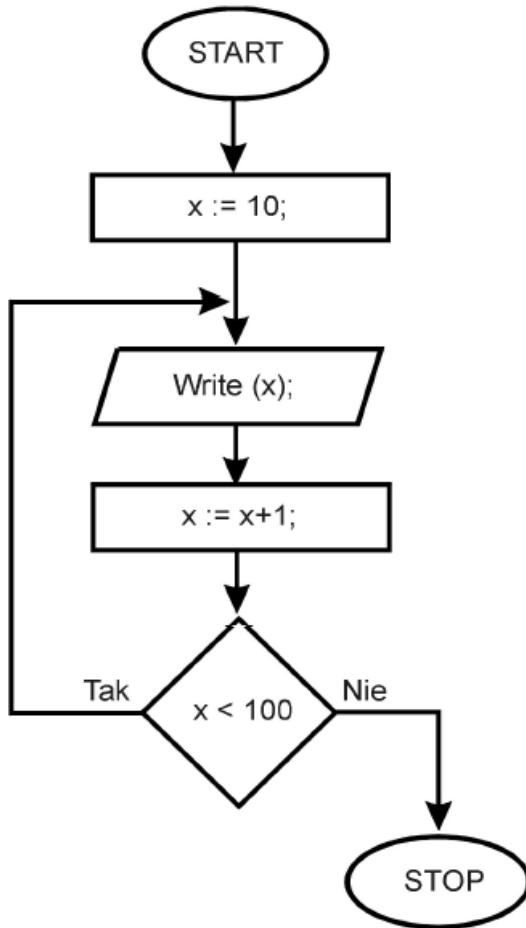
8



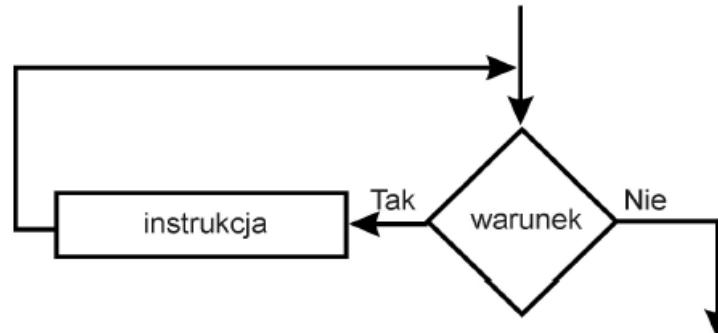
Warunek musi być tak określony, aby jego ocena prawdziwości była jednoznaczna.

Instrukcja iteracji

9



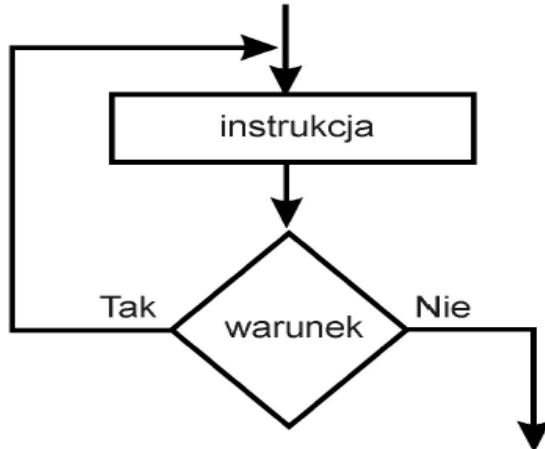
Poniżej przedstawiamy trzy podstawowe przypadki iteracji stosowanych w algorytmach.



Najpierw sprawdzany jest warunek, potem wykonywana jest instrukcja. (dopóki spełniony jest warunek wykonuj instrukcję - w Pascalu : While warunek do instrukcja)

Instrukcja iteracji

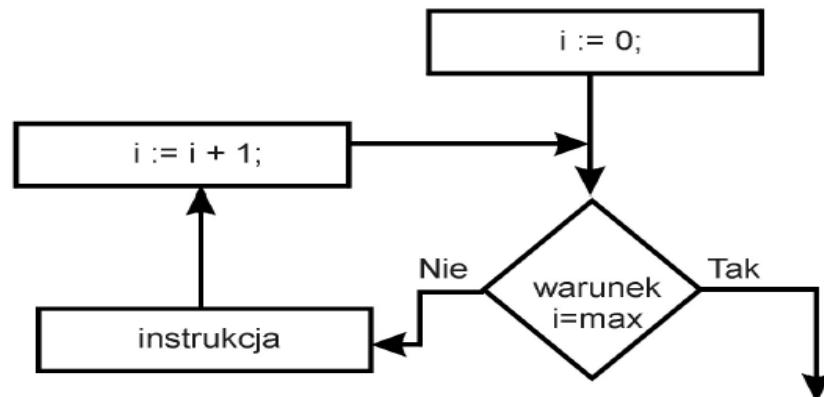
10



Najpierw wykonywana jest instrukcja , a potem sprawdzany jest warunek (wykonuj instrukcję dopóki spełniony jest warunek – w Pascalu : repeat instrukcja until wyrażenie).

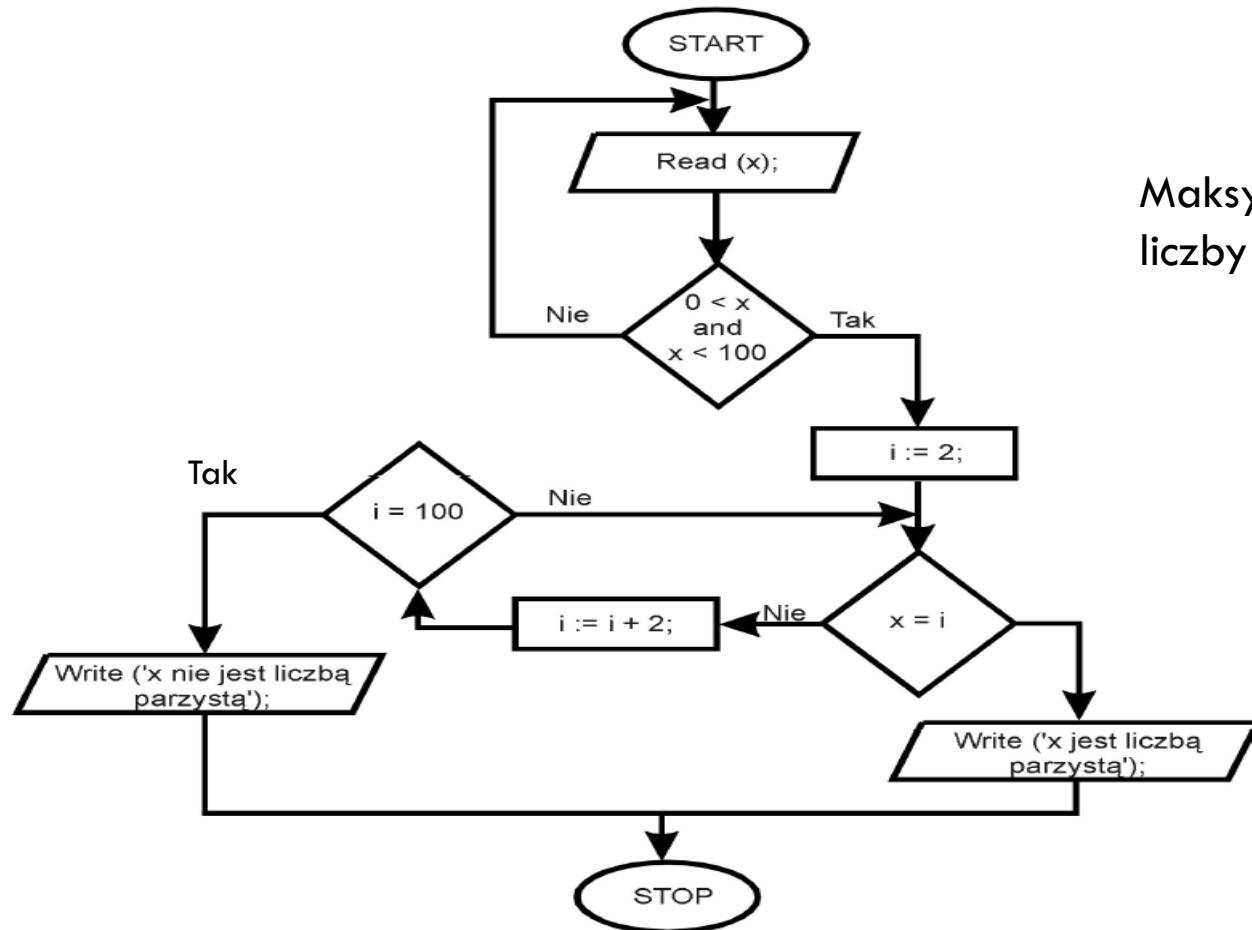
Instrukcja wykonywana jest z góry ustaloną (max) ilość razy (w Pascalu : For zmienna := wyrażenie_1 to max do (downto) instrukcja)

W pętli tej wartość licznika „i” zwiększana jest o 1 po każdej wykonywanej instrukcji czyli jest **inkrementowana**



Badanie parzystości: algorytm 1

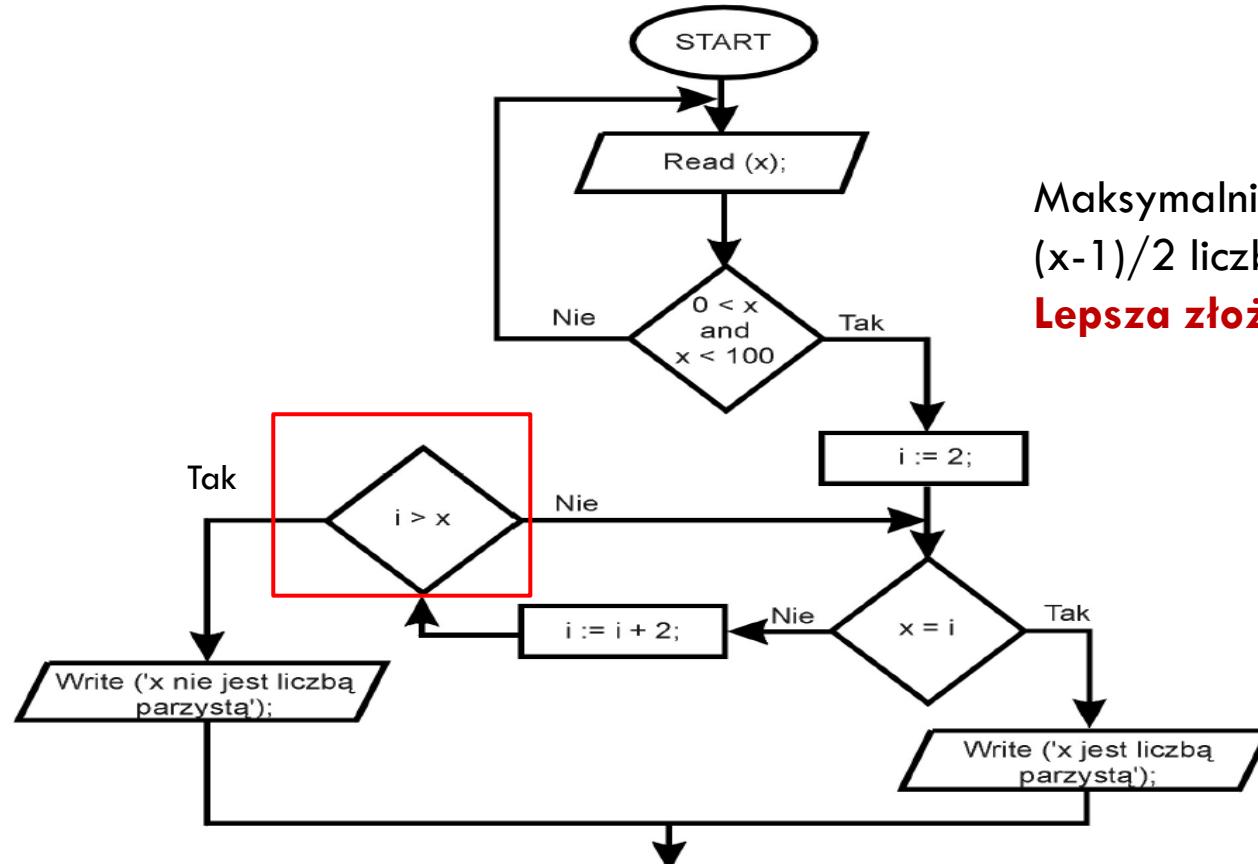
11



Maksymalnie 49 porównań
liczby i z liczbą 100

Badanie parzystości: algorytm 2

12



Maksymalnie porównań
 $(x-1)/2$ liczby i z liczbą x.
Lepsza złożoność obliczeniowa

Algorytm Euklidesa

13

□ Największy wspólny dzielnik dwóch liczb.

Pobieramy dwie liczby naturalne, od większej z nich odejmujemy mniejszą, a następnie większą liczbę zastępujemy różnicą. Postępujemy tak do momentu, gdy dwie liczby będą równe. Otrzymana liczba będzie NWD.

Przykład : $a = 12$, $b = 20$, ponieważ $b > a$ to $b = 20 - 12 = 8$, teraz $a > b$ czyli $a = 12 - 8 = 4$, dalej $b > a$ czyli $b = 8 - 4 = 4$ i $a = 4$ stąd NWD = 4

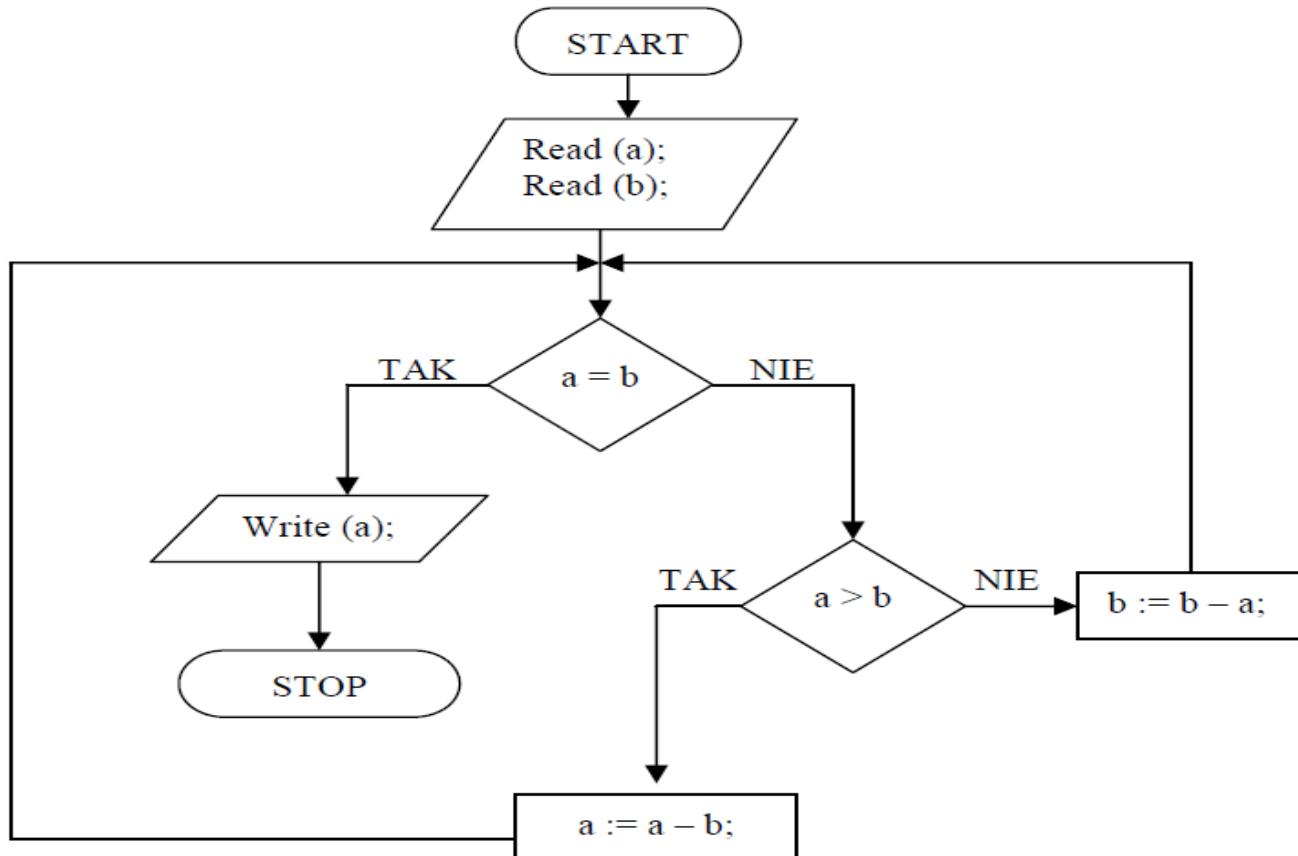
Równie często stosuje się modyfikację algorytmu $\text{NWD}(a,b) = \text{NWD}(a \bmod b, b - (a \bmod b))$

Będziemy iteracyjnie zmieniać wartości a i b aż do momentu, gdy a osiągnie wartość 0.

Przykład : $a = 12$, $b = 20$, $a = 12 \bmod 20 = 12$, $b = 20 - 12 = 8$ następnie $a = 12 \bmod 8 = 4$, $b = 8 - 4 = 4$, i następny krok $a = 4 \bmod 4 = 0$ czyli $b = \text{NWD} = 4$

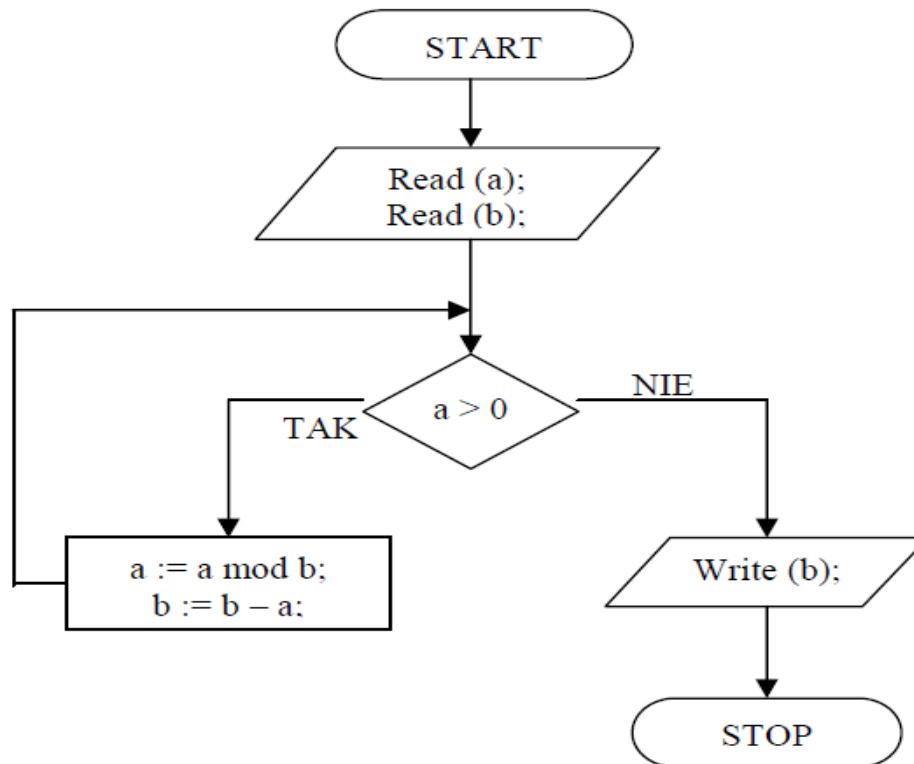
Algorytm Euklidesa: wersja 1

14



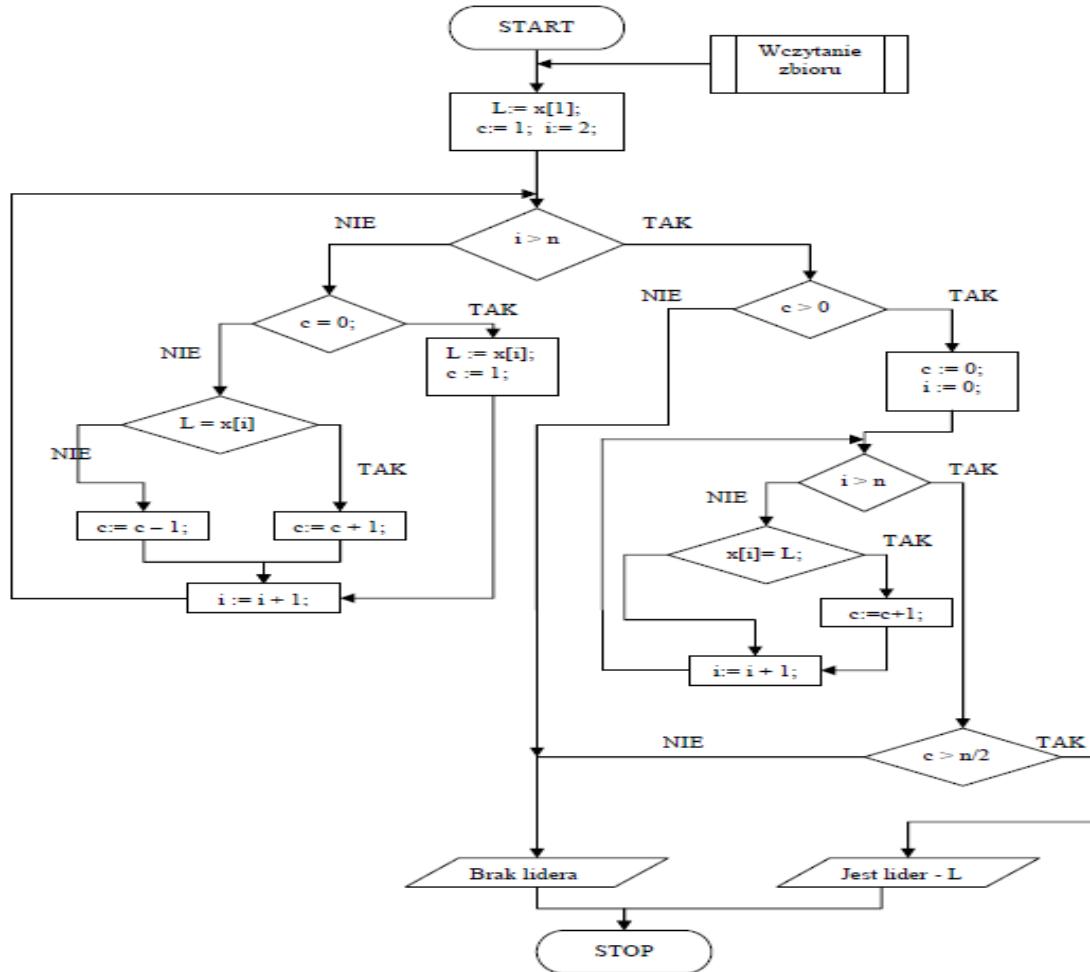
Algorytm Euklidesa: wersja 2

15



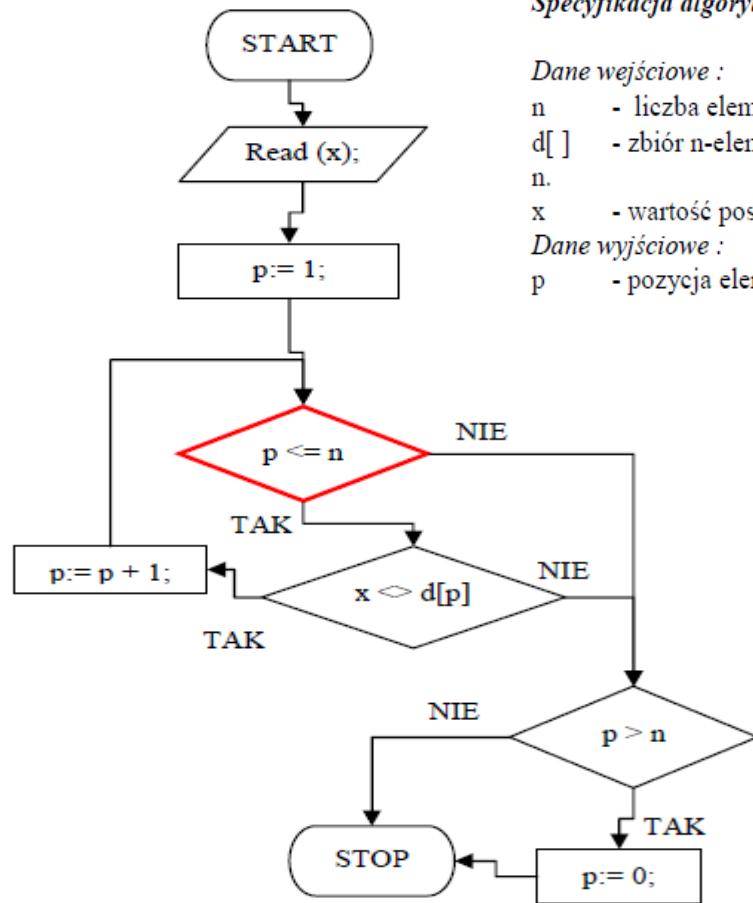
Poszukiwanie lidera zbioru

16



Przeszukiwanie sekwencyjne

17



Specyfikacja algorytmu :

Dane wejściowe :

n - liczba elementów w sortowanym zbiorze, $n \in N$

$d[]$ - zbiór n -elementowy, który będzie przeszukiwany. Elementy zbioru mają indeksy od 1 do n .

x - wartość poszukiwana

Dane wyjściowe :

p - pozycja elementu x w zbiorze $d[]$. Jeśli $p = 0$, to element x w zbiorze nie występuje.

**Warunek gwarantuje zakończenie pętli,
możemy też wprowadzić wartownika**

Złożoność obliczeniowa $O(n)$

Poszukiwanie najczęstszego elementu występującego w zbiorze

18

Specyfikacja problemu

Dane wejściowe :

n - liczba elementów w zbiorze wejściowym

d[] - zbiór wejściowy, w którym dokonujemy poszukiwań. Indeksy elementów rozpoczynają się od 1. Zbiór musi posiadać miejsce na dodatkowy element, który zostanie dopisany na końcu.

Dane wyjściowe :

w_n - wartość elementu powtarzającego się najczęściej

p_n - pierwsza pozycja elementu najczęstszego

L_n - liczba wystąpień najczęstszego elementu

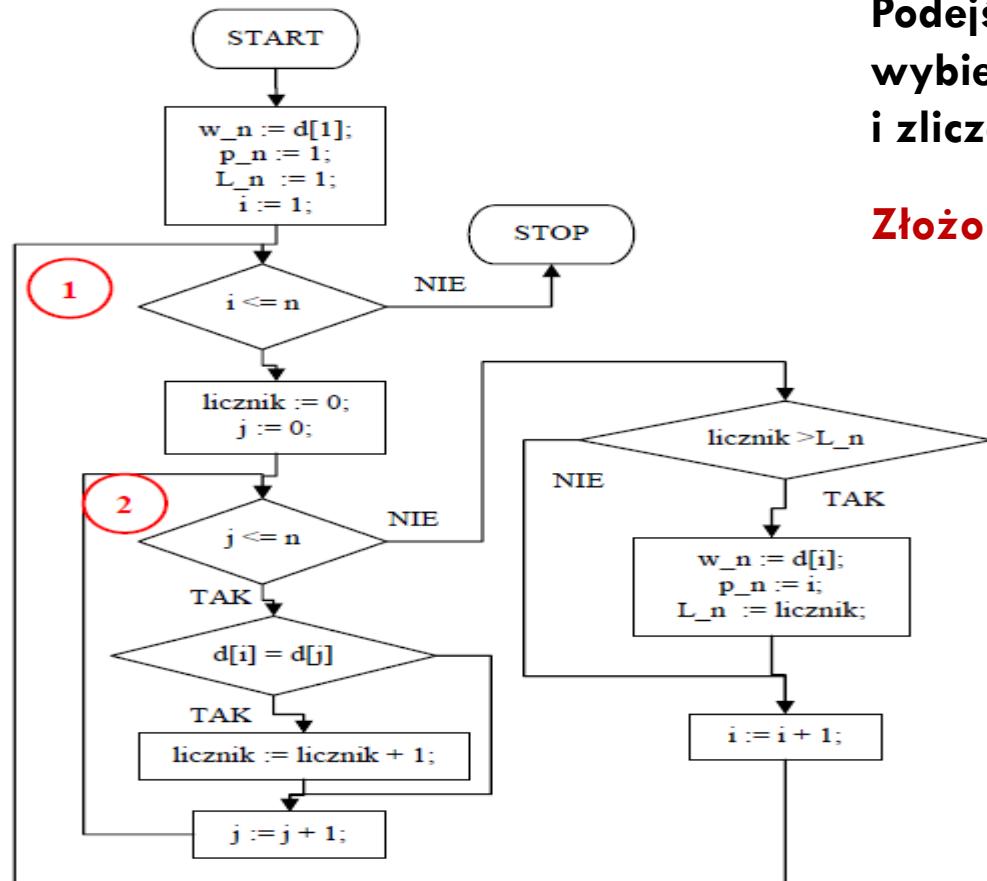
Zmienne pomocnicze :

i,j - zmienne licznikowe pętli

licznik - licznik wystąpień elementu

Poszukiwanie najczęstszego elementu występującego w zbiorze

19

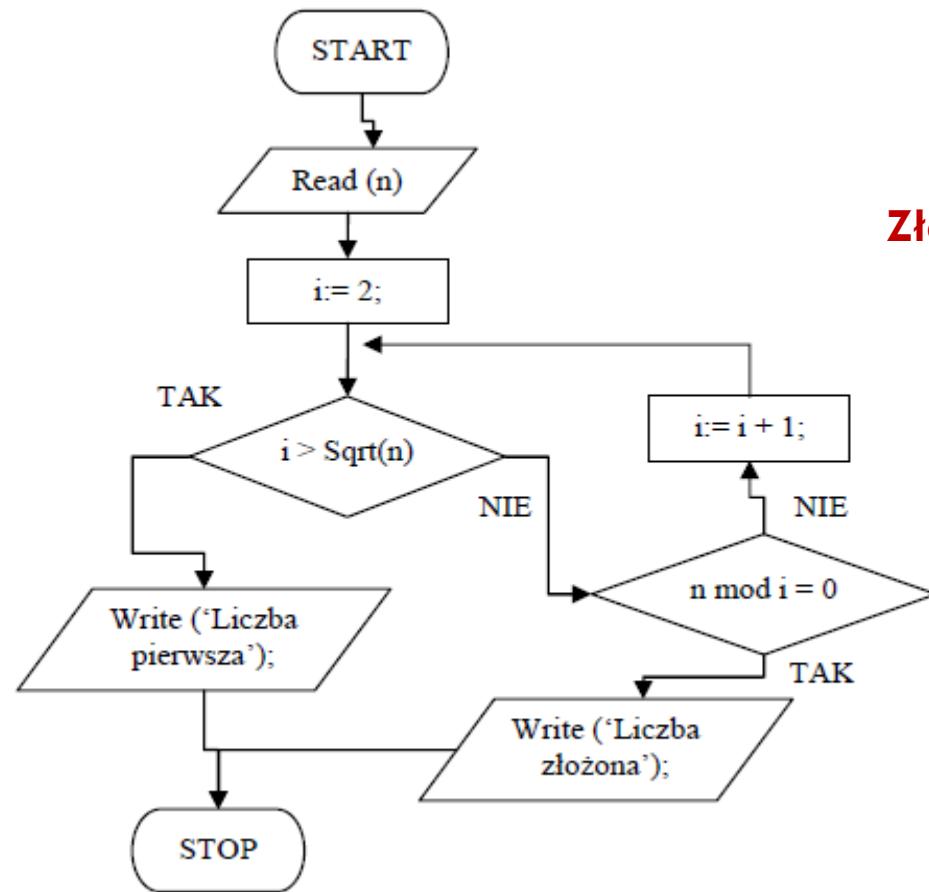


Podejście bezpośrednie:
wybieramy kolejne elementy zbioru
i zliczamy częstotliwość ich występowania.

Złożoność obliczeniowa $O(n^2)$

Algorytm sprawdzający czy liczba jest liczbą pierwszą.

20



Złożoność obliczeniowa $O(n^{1/2})$

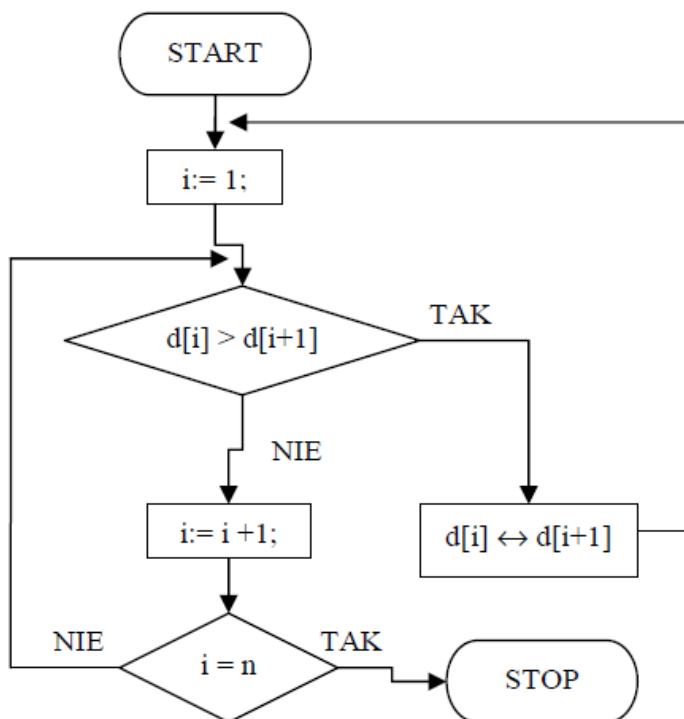
Złożoność obliczeniowa

21

Porównanie klas złożoności obliczeniowych		
Klasa złożoności obliczeniowej	Nazwa klasy złożoności obliczeniowej	Cechy algorytmu
$\Theta(1)$	stała	działa prawie natychmiast
$\Theta(\log n)$	logarytmiczna	bardzo szybki
$\Theta(n)$	liniowa	szybki
$\Theta(n \log n)$	liniowo-logarytmiczna	dosyć szybki
$\Theta(n^2)$	kwadratowa	wolny dla dużych n
$\Theta(n^3)$	sześcienna	wolny dla większych n
$\Theta(2^n), \Theta(n!)$	wykładnicza	nierozwiązalny dla większych n

Sortowanie naiwne

22



Cechy Algorytmu Sortowania Naiwnego	
klasa złożoności obliczeniowej optymistyczna	$\Theta(n) - \Theta(n^2)$
klasa złożoności obliczeniowej typowa	$\Theta(n^3)$
klasa złożoności obliczeniowej pesymistyczna	$\Theta(n^3)$
Sortowanie w miejscu	TAK
Stabilność	TAK

Pesymistyczna:

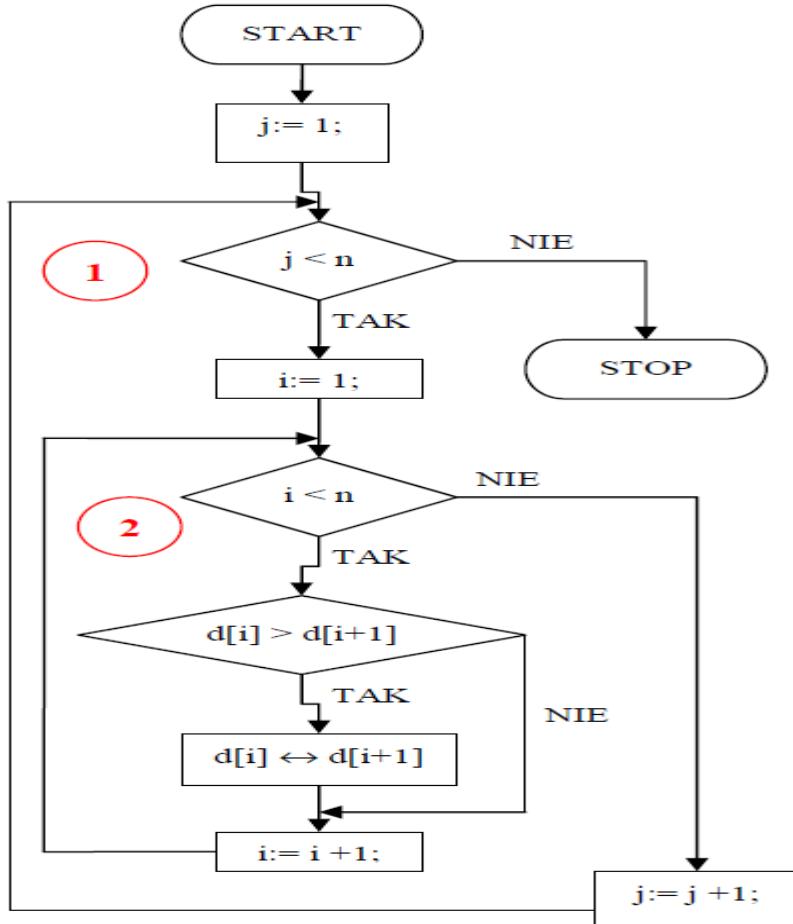
dla zbiorów posortowanych odwrotnie

Optymistyczna:

dla zbiorów uporządkowanych z niewielką ilością elementów nie na swoich miejscach

Sortowanie bąbelkowe

23



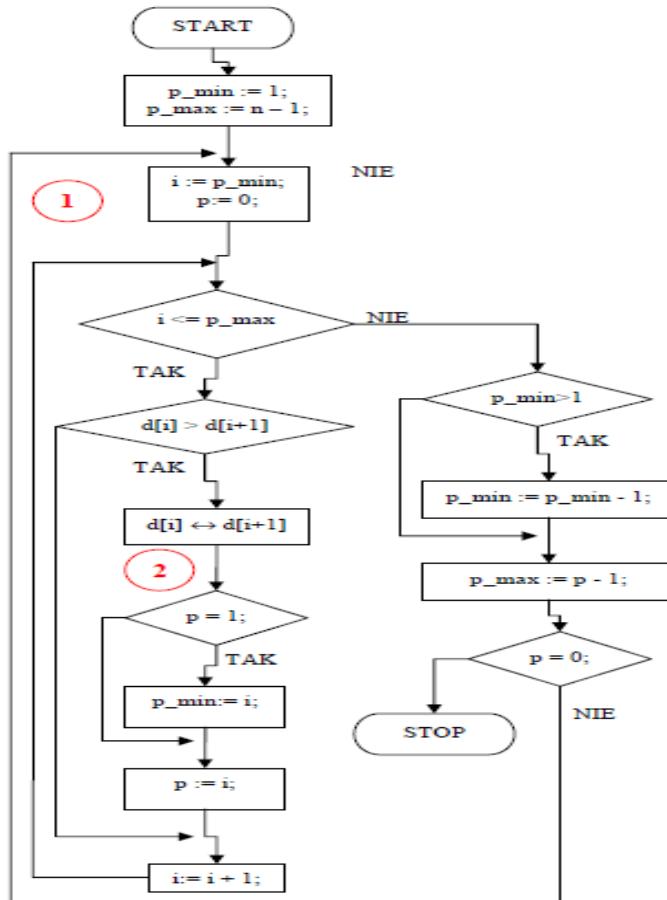
Sortowanie wykonywane jest w dwóch zagnieżdżonych pętlach. Pętla zewnętrzna nr 1 kontrolowana jest przez zmienną j . Wykonuje się ona $n - 1$ razy. Wewnątrz pętli nr 1 umieszczona jest pętla nr 2 sterowana przez zmienną i . Wykonuje się ona również $n - 1$ razy. W efekcie algorytm wykonuje w sumie: $T_1(n) = (n - 1)^2 = n^2 - 2n + 1$ obiegów pętli wewnętrznej, po których zakończeniu zbiór zostanie posortowany.

Cechy Algorytmu Sortowania Bąbelkowego wersja nr 1

klasa złożoności obliczeniowej optymistyczna	$\Theta(n^2)$
klasa złożoności obliczeniowej typowa	$\Theta(n^2)$
klasa złożoności obliczeniowej pesymistyczna	$\Theta(n^2)$
Sortowanie w miejscu	TAK
Stabilność	TAK

Sortowanie bąbelkowe: modyfikacje

24

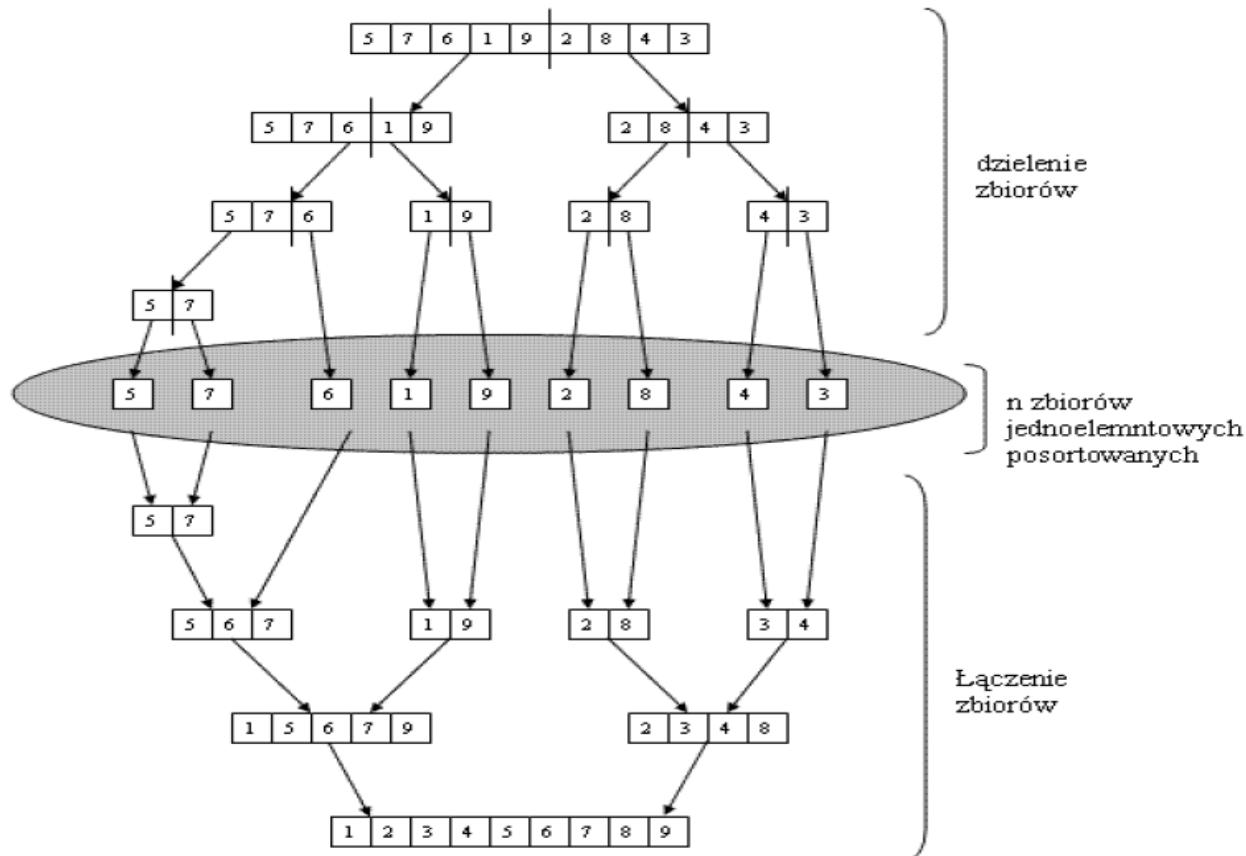


Cechy Algorytmu Sortowania Bąbelkowego
wersja nr 2

klasa złożoności obliczeniowej optymistyczna	$\Theta(n)$
klasa złożoności obliczeniowej typowa	$\Theta(n^2)$
klasa złożoności obliczeniowej pesymistyczna	$\Theta(n^2)$
Sortowanie w miejscu	TAK
Stabilność	TAK

Rekurencja: sortowanie

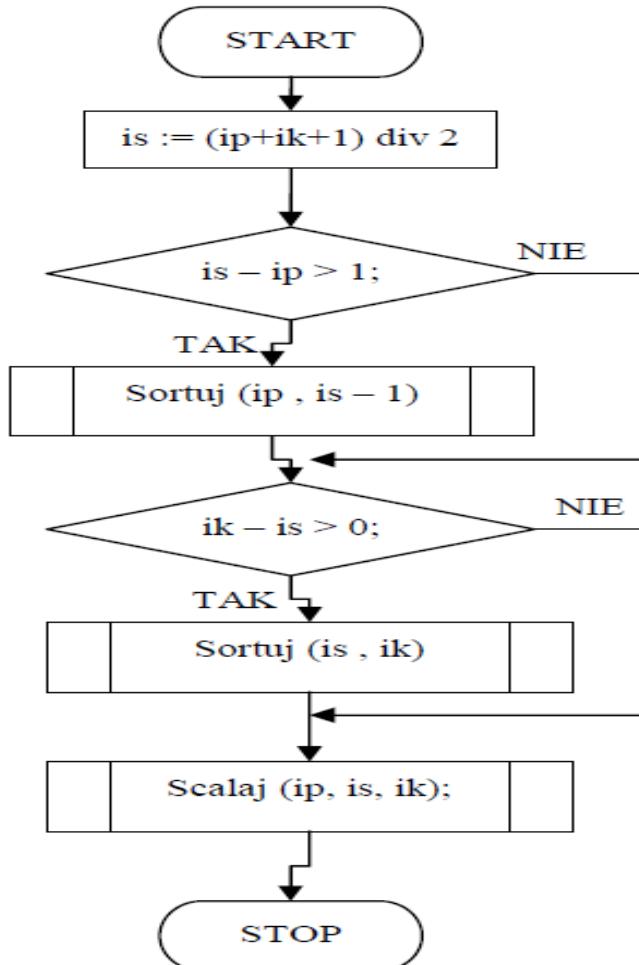
25



Przykład : sortujemy zbiór o postaci: { 6 5 4 1 3 7 9 2 }

Rekurencja: program sortuj

26



Złożoność obliczeniowa

$O(n \log(n))$

Dane wejściowe :

- d [] - zbiór scalany
ip - indeks pierwszego elementu w młodszym podzbiorze, $ip \in \mathbb{N}$
ik - indeks ostatniego elementu w starszym podzbiorze, $ik \in \mathbb{N}$

Dane wyjściowe :

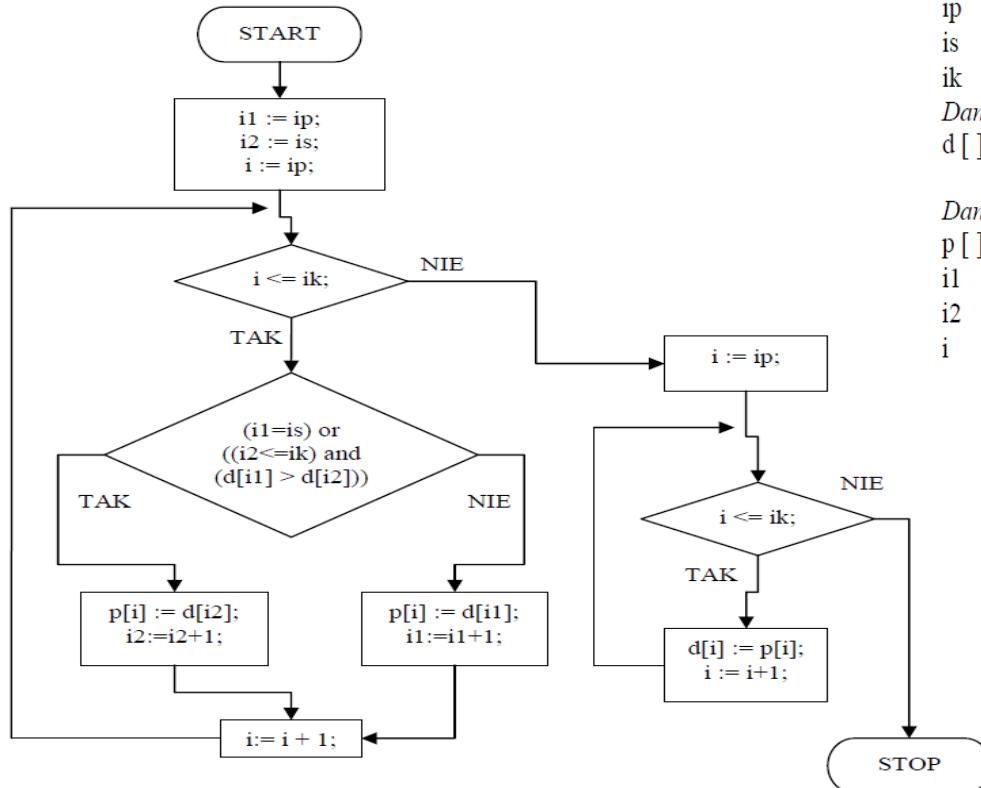
- d [] - zbiór scalony

Dane pomocnicze :

- is - indeks pierwszego elementu w starszym podzbiorze, $is \in \mathbb{N}$

Rekurencja: blok scalaj

27



Dane wejściowe :

- $d[]$ - zbiór scalany
 ip - indeks pierwszego elementu w młodszym podzbiorze, $ip \in N$
 is - indeks pierwszego elementu w starszym podzbiorze, $is \in N$
 ik - indeks ostatniego elementu w starszym podzbiorze, $ik \in N$

Dane wyjściowe :

- $d[]$ - zbiór scalony

Dane pomocnicze :

- $p[]$ - zbiór pomocniczy zawierający tyle samo elementów ile zbiór d
 $i1$ - indeks elementów w młodziej połówce zbioru $d[]$, $i1 \in N$
 $i2$ - indeks elementów w starszej połówce zbioru $d[]$, $i2 \in N$
 i - indeks elementów w zbiorze $p[]$, $i \in N$

Złożoność obliczeniowa $O(n)$

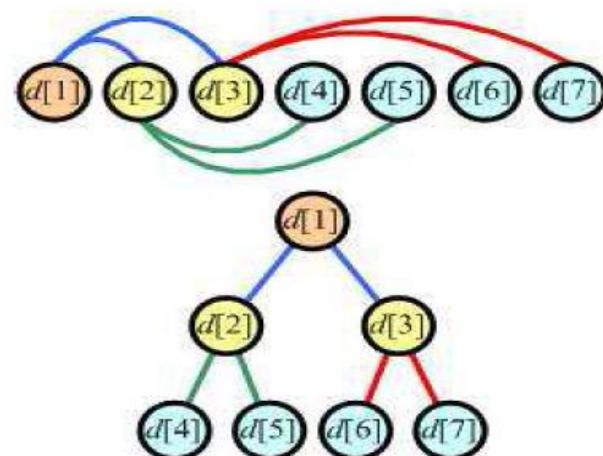
Sortowanie stogowe

28

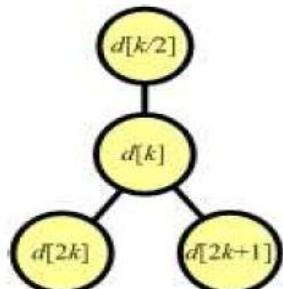
Zastosujmy następujące odwzorowanie:

- Element $d[1]$ będzie zawsze korzeniem drzewa.
- i -ty poziom drzewa binarnego wymaga 2^{i-1} węzłów. Będziemy je kolejno pobierać z tablicy.

Otrzymamy w ten sposób następujące odwzorowanie elementów tablicy w drzewo binarne:



Dla węzła k -tego wyprowadzamy następujące wzory:

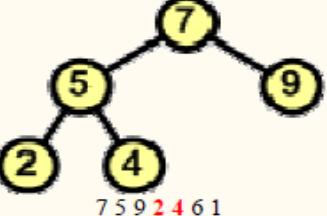
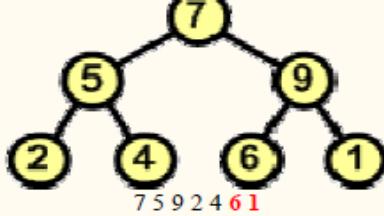


węzły potomne mają indeksy równe:
 $2k$ - lewy potomek
 $2k+1$ - prawy potomek
węzeł nadrzędny ma indeks równy $\lceil k / 2 \rceil$ (dzielenie całkowitoliczbowe)

Drzewo binarne

29

Przykład : Skonstruować drzewo binarne z elementów zbioru {7 5 9 2 4 6 1}

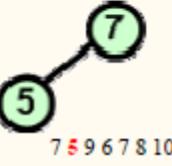
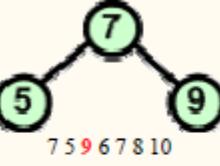
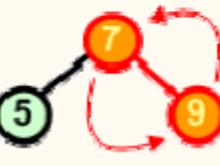
Operacja	Opis
	Konstrukcję drzewa binarnego rozpoczynamy od korzenia, który jest pierwszym elementem zbioru, czyli liczbą 7.
	Do korzenia dołączamy dwa węzły potomne, które leżą obok w zbiorze. Są to dwa kolejne elementy, 5 i 9.
	Do lewego węzła potomnego (5) dołączamy jego węzły potomne. Są to kolejne liczby w zbiorze, czyli 2 i 4.
	Pozostaje nam dołączyć do prawego węzła ostatnie dwa elementy zbioru, czyli liczby 6 i 1. Drzewo jest kompletne.

Kopiec : tworzenie

30

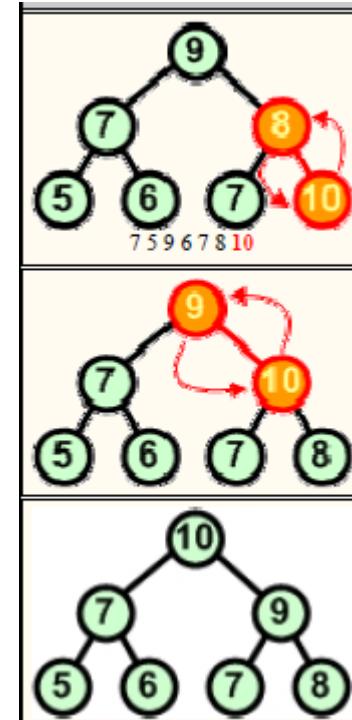
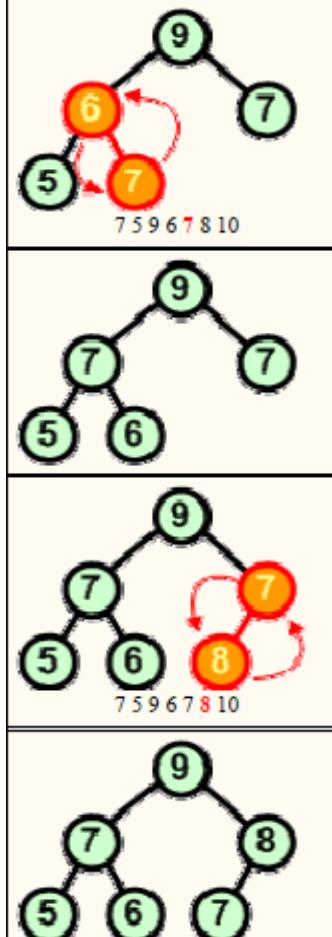
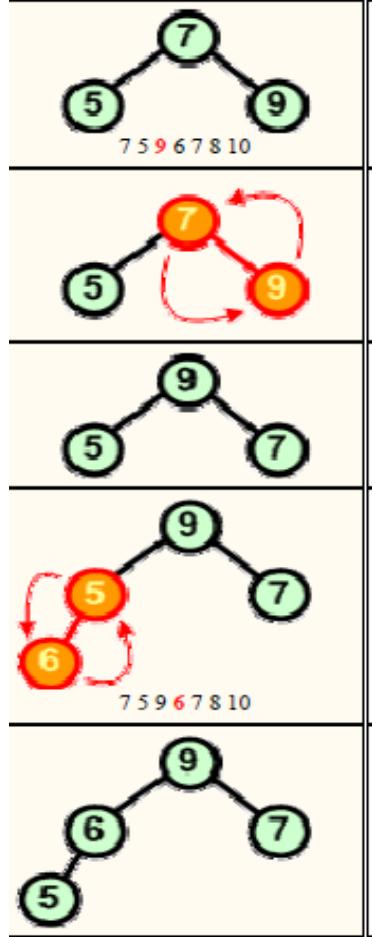
Kopiec jest drzewem binarnym, w którym wszystkie węzły spełniają następujący warunek (zwany warunkiem kopca) : węzeł nadzędny jest większy lub równy węzłom potomnym (w porządku malejącym relacja jest odwrotna - mniejszy lub równy).

Przykład : Skonstruować kopiec z elementów zbioru {7 5 9 6 7 8 10}

Operacja	Opis
	Budowę kopca rozpoczynamy od pierwszego elementu zbioru, który staje się korzeniem.
	Do korzenia dołączamy następny element. Warunek kopca jest zachowany.
	Dodajemy kolejny element ze zbioru.
	Po dodaniu elementu 9 warunek kopca przestaje być spełniony. Musimy go przywrócić. W tym celu za nowy węzeł nadzędny wybieramy nowo dodany węzeł. Poprzedni węzeł nadzędny wędruje w miejsce węzła dodanego - zamieniamy węzły 7 i 9 miejscami.

Kopiec

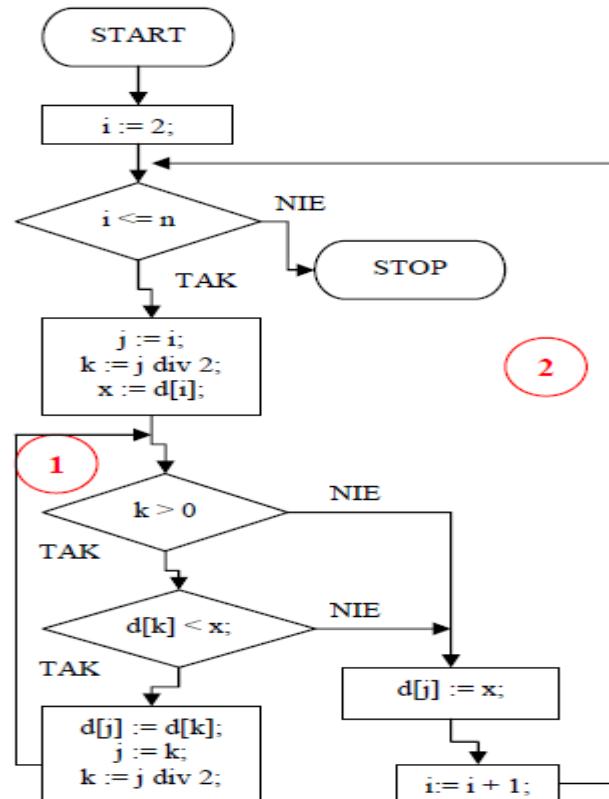
31



Kopiec

32

Schemat blokowy



Złożoność obliczeniowa $O(n \log(n))$

Prof. dr hab. Elżbieta Richter-Wąs

Algorytm tworzy kopiec w tym samym zbiorze wejściowym $d[]$. Nie wymaga zatem dodatkowych struktur danych i ma złożoność pamięciową klasy $\Theta(n)$.

Pętla nr 1 wyznacza kolejne elementy wstawiane do kopca. Pierwszy element pomijamy, ponieważ zostałby i tak na swoim miejscu. Dlatego pętla rozpoczyna wstawianie od elementu nr 2.

Wewnątrz pętli nr 1 inicjujemy kilka zmiennych:
j - pozycja wstawianego elementu (liścia)
k - pozycja elementu nadzawanego (przodka)
x - zapamiętuje wstawiany element

Następnie rozpoczynamy pętlę warunkową nr 2, której zadaniem jest znalezienie w kopcu miejsca do wstawienia zapamiętanego elementu w zmiennej x. Pętla ta wykonuje się do momentu osiągnięcia korzenia kopca ($k = 0$) lub znalezienia przodka większego od zapamiętanego elementu. Wewnątrz pętli przesuwamy przodka na miejsce potomka, aby zachować warunek kopca, a następnie przesuwamy pozycję j na pozycję zajmowaną wcześniej przez przodka. Pozycja k staje się pozycją nowego przodka i pętla się kontynuuje. Po jej zakończeniu w zmiennej j znajduje się numer pozycji w zbiorze $d[]$, na której należy umieścić element w x.

Po zakończeniu pętli nr 1 w zbiorze zostaje utworzona struktura kopca.

9/01/2017

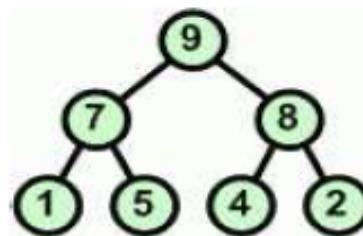
Kopiec: rozbieranie

33

Zasady rozbioru kopca :

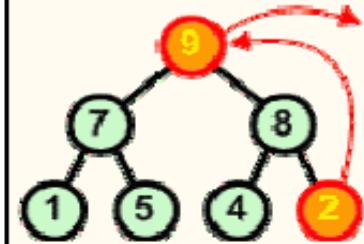
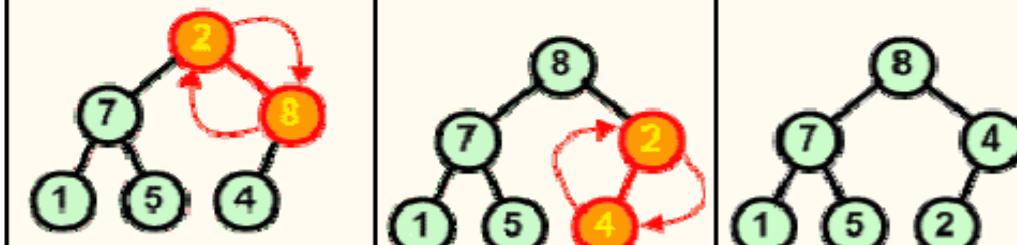
1. Zamień miejscami korzeń z ostatnim liściem, który wylącz ze struktury kopca. Elementem pobieranym z kopca jest zawsze jego korzeń, czyli element największy.
2. Jeśli jest to konieczne, przywróć warunek kopca idąc od korzenia w dół.
3. Kontynuuj od kroku 1, aż kopiec będzie pusty.

Przykład : Rozebrać kopiec



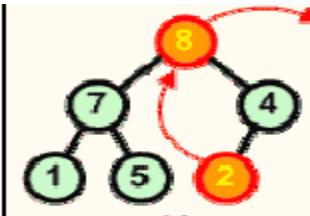
Kopiec: rozbieranie

34

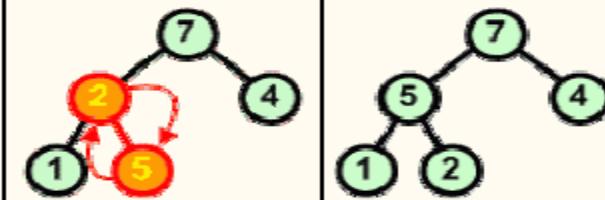
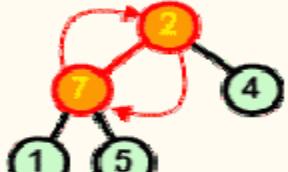
Z	Operacja	Opis
1		Rozbiór kopca rozpoczynamy od korzenia, który usuwamy ze struktury kopca. W miejsce korzenia wstawiamy ostatni liść.
P		Poprzednia operacja zaburzyła strukturę kopca. Idziemy zatem od korzenia w dół struktury przywracając warunek kopca - przodek większy lub równy od swoich potomków. Praktycznie polega to na zamianie przodka z największym potomkiem. Operację kontynuujemy dotąd, aż natrafimy na węzły spełniające warunek kopca.

Kopiec: rozbieranie

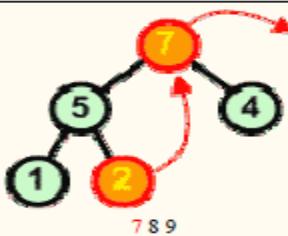
35



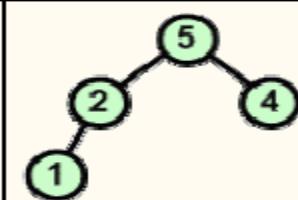
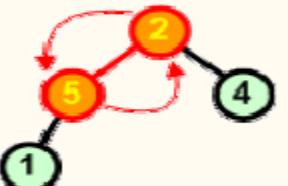
Usuwamy z kopca kolejny korzeń zastępując go ostatnim liściem



W nowym kopcu przywracamy warunek kopca.



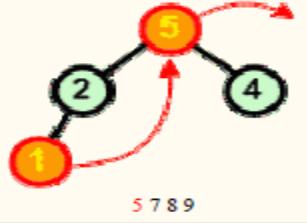
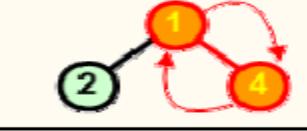
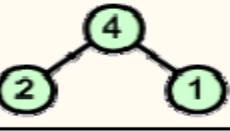
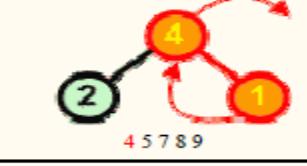
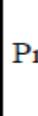
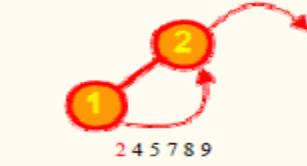
Usuwamy z kopca kolejny korzeń zastępując go ostatnim liściem



W nowym kopcu przywracamy warunek kopca. W tym przypadku już po pierwszej wymianie węzłów warunek koca jest zachowany w całej strukturze.

Kopiec: rozbieranie

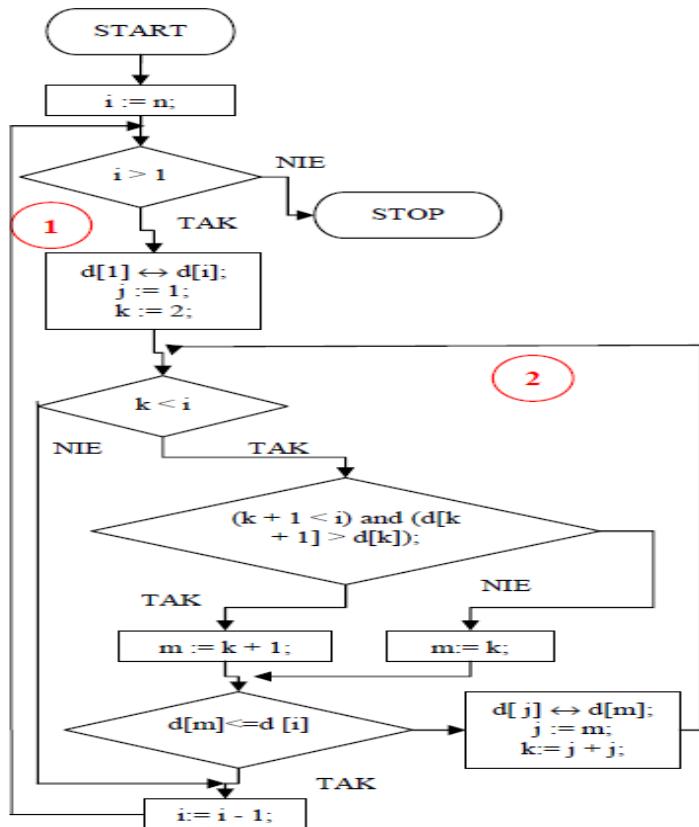
36

	Usuwamy z kopca kolejny korzeń zastępując go ostatnim liściem
	 Przywracamy warunek kopca w strukturze.
	Usuwamy z kopca kolejny korzeń zastępując go ostatnim liściem
	 Przywracamy warunek kopca w strukturze.
	Usuwamy z kopca kolejny korzeń zastępując go ostatnim liściem.
	Po wykonaniu poprzedniej operacji usunięcia w kopcu pozostał tylko jeden element - usuwamy go. Zwróć uwagę, iż usunięte z kopca elementy tworzą ciąg uporządkowany.

Kopiec: rozbieranie

37

Schemat blokowy



Rozbiór kopca wykonywany jest w dwóch zagnieżdżonych pętlach. Pętla nr 1 zamienia miejscami kolejne liście ze spodu drzewa z korzeniem. Zadaniem pętli nr 2 jest przywrócenie w strukturze warunku kopca.

**Złożoność obliczeniowa
 $O(n \log(n))$**

Sortowanie przez kopcowanie

38

- Krok 1 : Tworz_Kopiec
- Krok 2 : Rozbierz_Kopiec
- Krok 3 : Zakończ algorytm

Cechy Algorytmu Sortowania Przez Kopcowanie	
klasa złożoności obliczeniowej optymistyczna	
klasa złożoności obliczeniowej typowa	$\Theta(n \log n)$
klasa złożoności obliczeniowej pesymistyczna	
Sortowanie w miejscu	TAK
Stabilność	NIE

Ponieważ sortowanie przez kopcowanie składa się z dwóch następujących bezpośrednio po sobie operacji o klasie czasowej złożoności obliczeniowej $\Theta(n \log n)$, to dla całego algorytmu klasa złożoności również będzie wynosić $\Theta(n \log n)$.

Algorytmy sortujące

39

Nazwa algorytmu sortującego	Klasa złożoności			Stabilność	Sortowanie w miejscu	Zalecane?
	optymistyczna	typowa	pesymistyczna			
Zwariowane	$\Theta(1)$	$\Theta(n*n!)$	$\Theta(\infty)$	NIE	TAK	NIE!!!
Naiwne	$\Theta(n) \dots \Theta(n^2)$	$\Theta(n^3)$	$\Theta(n^3)$	TAK	TAK	NIE
Bąbelkowe wersja 1	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	TAK	TAK	NIE
Bąbelkowe wersja 2	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	TAK	TAK	TAK
Przez wybór	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	NIE	TAK	TAK/NIE
Przez wstawianie	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	TAK	TAK	TAK
Metodą Shella	$\Theta(n^{1,14})$	$\Theta(n^{1,15})$	$\Theta(n^{1,15})$	NIE	TAK	TAK
Przez łączenie	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	TAK	NIE	TAK
Przez kopcowanie	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	NIE	TAK	TAK
Szybkie	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n^2)$	NIE	TAK	TAK
Kubelkowe wersja I	$\Theta(m + n)$	$\Theta(m + n)$	$\Theta(m + n)$	NIE	NIE	TAK/NIE
Radix Sort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	TAK	NIE	TAK

Algorytmy i Struktury Danych.

Grafy. Drzewo rozpinające.

dr hab. Bożena Woźna-Szcześniak
bwozna@gmail.com

Jan Długosz University, Poland

Wykład 11



Na podstawie materiałów prof. dr. hab. Michała Karońskiego do wykładu z algorytmów grafowych (AGR320).

Plan wykładu

- Drzewo rozpinające

Drzewo rozpinające grafu

Definicja

Drzewem rozpinającym (rozpięтыm) grafu G nazywamy spójny i acykliczny podgraf grafu G zawierający wszystkie jego wierzchołki.

Drzewo rozpinające grafu

Definicja

Drzewem rozpinającym (rozpięтыm) grafu G nazywamy spójny i acykliczny podgraf grafu G zawierający wszystkie jego wierzchołki.

Twierdzenie

Każdy graf spójny zawiera drzewo rozpinające.

Drzewo rozpinające grafu

Definicja

Drzewem rozpinającym (rozpięтыm) grafu G nazywamy spójny i acykliczny podgraf grafu G zawierający wszystkie jego wierzchołki.

Twierdzenie

Każdy graf spójny zawiera drzewo rozpinające.

Twierdzenie

W grafie spójnym $G = (V, E)$ krawędź $e \in E$ jest krawędzią cięcia wtedy i tylko wtedy, gdy e należy do każdego drzewa rozpinającego grafu G .

Drzewo rozpinające grafu

- Generowanie drzewa rozpinającego danego grafu spójnego G na n wierzchołkach można realizować biorąc kolejno krawędzie z pewnej listy i akceptując je po sprawdzeniu, czy nie tworzą one cyklu z dotychczas zaakceptowanymi krawędziami.

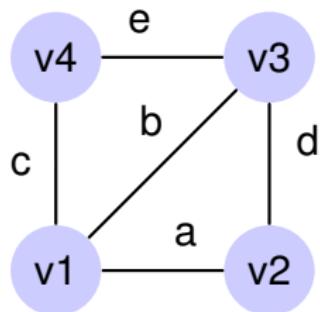
Drzewo rozpinające grafu

- Generowanie drzewa rozpinającego danego grafu spójnego G na n wierzchołkach można realizować biorąc kolejno krawędzie z pewnej listy i akceptując je po sprawdzeniu, czy nie tworzą one cyklu z dotychczas zaakceptowanymi krawędziami.
- Generowanie wszystkich drzew rozpinających danego grafu jest złożone obliczeniowo, ponieważ drzew rozpinających może być bardzo dużo.
- Rozpoczniemy: uporządkujmy wszystkie krawędzie grafu w dowolny sposób tworząc listę krawędzi (e_1, e_2, \dots, e_m) , gdzie $m = |E|$.
- Każdy podzbiór krawędzi będziemy przedstawiać w postaci listy uporządkowanej, zgodnie z porządkiem w liście krawędzi; do generowania wszystkich drzew rozpiętych wykorzystymy algorytm z nawrotami.

Drzewo rozpinające grafu - przykład

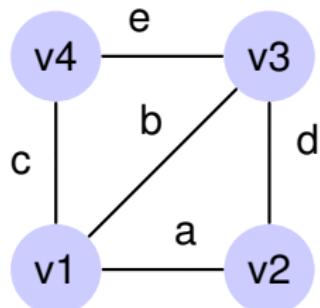
Znajdź wszystkie drzewa rozpinające grafu:

- Pokażemy, że graf ten zawiera 8 różnych drzew rozpinających.



Drzewo rozpinające grafu - przykład

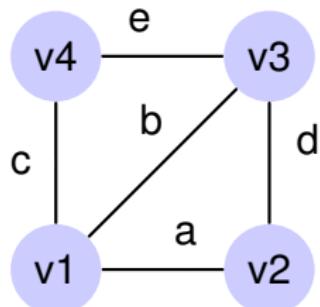
Znajdź wszystkie drzewa rozpinające grafu:



- Pokażemy, że graf ten zawiera 8 różnych drzew rozpinających.
- Łatwo zauważyc, że każde drzewo rozpinające tego grafu ma 3 krawędzie.

Drzewo rozpinające grafu - przykład

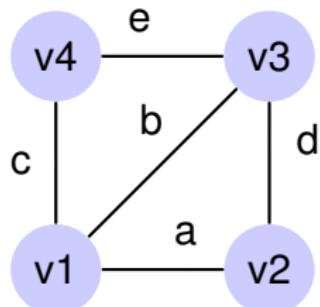
Znajdź wszystkie drzewa rozpinające grafu:



- Pokażemy, że graf ten zawiera 8 różnych drzew rozpinających.
- Łatwo zauważyc, że każde drzewo rozpinające tego grafu ma 3 krawędzie.
- Uporządkujmy wszystkie krawędzie grafu, tworząc listę krawędzi. Rozważać będziemy następującą listę krawędzi: **(a, b, c, d, e)**.

Drzewo rozpinające grafu - przykład

Znajdź wszystkie drzewa rozpinające grafu:

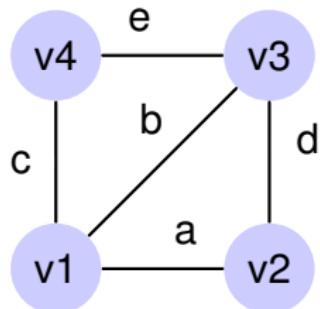


- Pokażemy, że graf ten zawiera 8 różnych drzew rozpinających.
- Łatwo zauważyc, że każde drzewo rozpinające tego grafu ma 3 krawędzie.
- Uporządkujmy wszystkie krawędzie grafu, tworząc listę krawędzi. Rozważać będziemy następującą listę krawędzi: **(a, b, c, d, e)**.

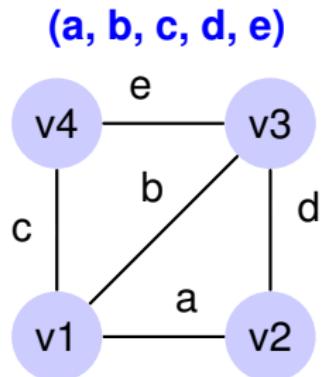
Drzewo rozpinające grafu - przykład

- Zgodnie z algorytmem z nawrotami, będziemy tworzyć kolejne listy.

(**a, b, c, d, e**)

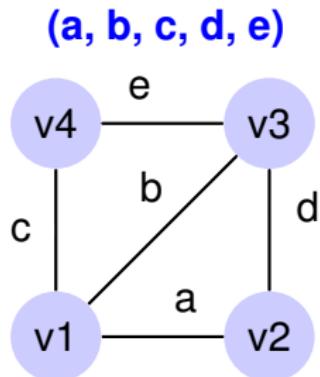


Drzewo rozpinające grafu - przykład



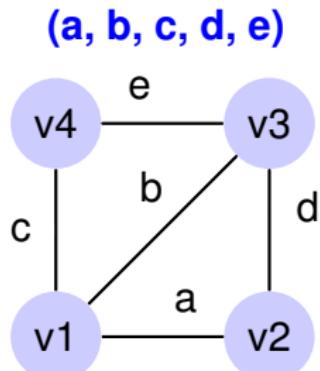
- Zgodnie z algorytmem z nawrotami, będziemy tworzyć kolejne listy.
- **lista czerwona** oznacza, że krawędzie te zawierają cykl i trzeba się wycofać.

Drzewo rozpinające grafu - przykład



- Zgodnie z algorytmem z nawrotami, będziemy tworzyć kolejne listy.
 - **lista czerwona** oznacza, że krawędzie te zawierają cykl i trzeba się wycofać.
 - lista podkreślona oznaczają drzewo rozpinające.

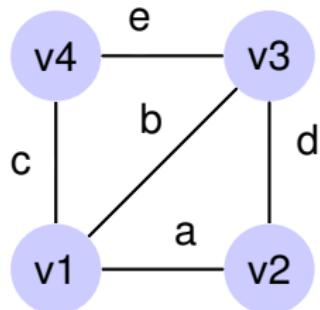
Drzewo rozpinające grafu - przykład



- Zgodnie z algorytmem z nawrotami, będziemy tworzyć kolejne listy.
 - **lista czerwona** oznacza, że krawędzie te zawierają cykl i trzeba się wycofać.
 - lista podkreślona oznaczają drzewo rozpinające.
- Listy (drzewa) generowane są w porządku leksykograficznym (alfabetycznym).

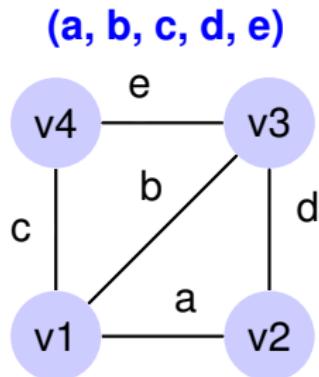
Drzewo rozpinające grafu - przykład

(**a, b, c, d, e**)



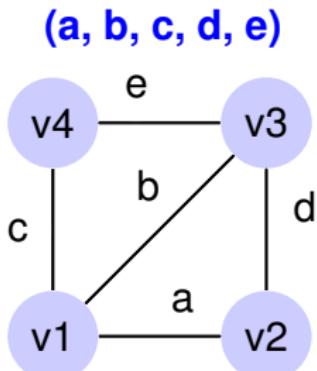
- (), (a), (a, b), (a, b, c) – Mamy pierwsze drzewo rozpinające.

Drzewo rozpinające grafu - przykład



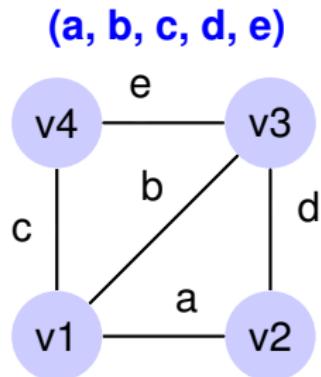
- $(), (a), (a, b), \underline{(a, b, c)}$ – Mamy pierwsze drzewo rozpinające.
- Usuwamy ostatnią krawędź z $\underline{(a, b, c)}$, dodajemy kolejną krawędź z listy i otrzymujemy: (a, b, d) - zawiera cykl.

Drzewo rozpinające grafu - przykład



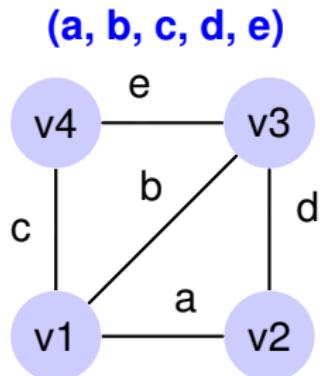
- (), (*a*), (*a, b*), (*a, b, c*) – Mamy pierwsze drzewo rozpinające.
- Usuwamy ostatnią krawędź z (*a, b, c*), dodajemy kolejną krawędź z listy i otrzymujemy: (*a, b, d*) - zawiera cykl.
- Usuwamy zatem *d* i generujemy następną listę: (*a, b, e*) – Otrzymaliśmy drugie drzewo rozpinające.

Drzewo rozpinające grafu - przykład



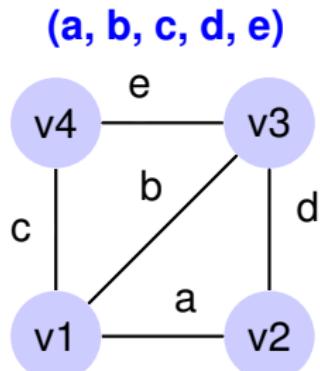
- Usuwamy krawędź e , nie możemy jednak kontynuować bo e jest ostatnią krawędzią na liście.

Drzewo rozpinające grafu - przykład



- Usuwamy krawędź e , nie możemy jednak kontynuować bo e jest ostatnią krawędzią na liście.
- Usuwamy kolejną krawędź - b i generujemy: (a, c) , (a, c, d) . Mamy kolejne drzewo rozpinające.

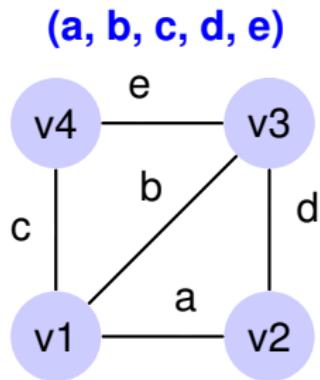
Drzewo rozpinające grafu - przykład



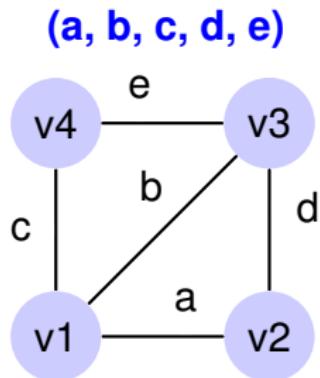
- Usuwamy krawędź e , nie możemy jednak kontynuować bo e jest ostatnią krawędzią na liście.
- Usuwamy kolejną krawędź - b i generujemy: (a, c) , (a, c, d) . Mamy kolejne drzewo rozpinające.
- Usuwamy ostatnią krawędź d i kontynuujemy: (a, c, e) . Znaleźliśmy kolejne drzewo rozpinające.

Drzewo rozpinające grafu - przykład

- Usuwamy krawędź e , następnie krawędź c , po czym generujemy kolejno: (a, d) , $\underline{(a, d, e)}$. Mamy kolejne drzewo rozpinające.

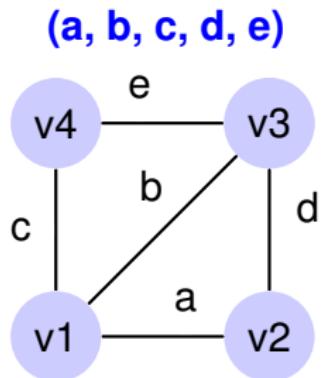


Drzewo rozpinające grafu - przykład



- Usuwamy krawędź e , następnie krawędź c , po czym generujemy kolejno: (a, d) , (a, d, e) . Mamy kolejne drzewo rozpinające.
- Usuwamy krawędź e , potem d tworzymy nową listę: (a, e) . Nie możemy jednak kontynuować więc usuwamy krawędzie e oraz a .

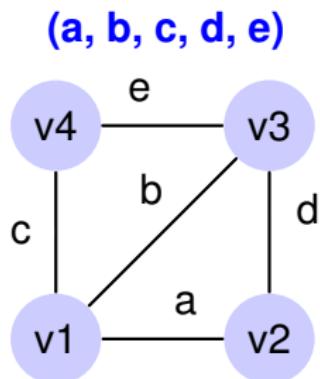
Drzewo rozpinające grafu - przykład



- Usuwamy krawędź e , następnie krawędź c , po czym generujemy kolejno: (a, d) , (a, d, e) . Mamy kolejne drzewo rozpinające.
- Usuwamy krawędź e , potem d tworzymy nową listę: (a, e) . Nie możemy jednak kontynuować więc usuwamy krawędzie e oraz a .
- Kolejnymi utworzonymi listami są więc: (b) , (b, c) , (b, c, d) . Otrzymaliśmy kolejne drzewo rozpinające.

Drzewo rozpinające grafu - przykład

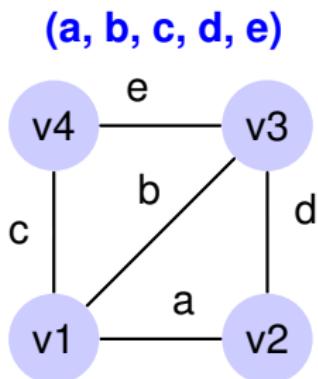
- Usuwamy krawędź d , ale kolejna lista: (b, c, e) zawiera cykl. Usuwamy więc kolejno krawędź e , potem c i generujemy listy (b, d) i $\underline{(b, d, e)}$. Znaleźliśmy kolejne drzewo rozpinające.



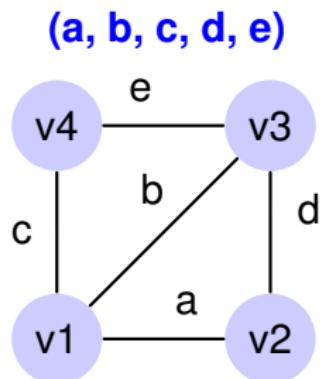
Drzewo rozpinające grafu - przykład

- Usuwamy krawędź d , ale kolejna lista: (b, c, e) zawiera cykl. Usuwamy więc kolejno krawędź e , potem c i generujemy listy (b, d) i $\underline{(b, d, e)}$. Znaleźliśmy kolejne drzewo rozpinające.

- Usuwamy kolejno krawędzie e i d a kolejną listą jest: (b, e) . Usuwamy e , a następnie b i tworzymy listy: (c) , (c, d) , $\underline{(c, d, e)}$. Mamy kolejne drzewo rozpinające.

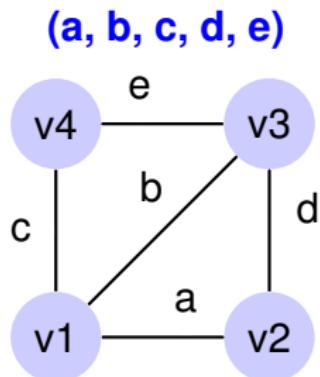


Drzewo rozpinające grafu - przykład



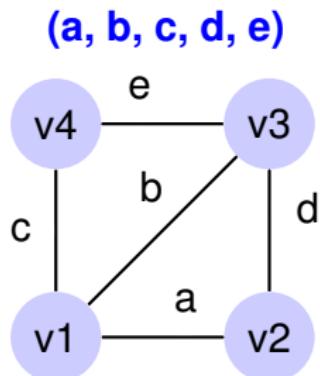
- Usuwamy krawędź d , ale kolejna lista: (b, c, e) zawiera cykl. Usuwamy więc kolejno krawędź e , potem c i generujemy listy (b, d) i $\underline{(b, d, e)}$. Znaleźliśmy kolejne drzewo rozpinające.
- Usuwamy kolejno krawędzie e i d a kolejną listą jest: (b, e) . Usuwamy e , a następnie b i tworzymy listy: (c) , (c, d) , $\underline{(c, d, e)}$. Mamy kolejne drzewo rozpinające.
- Usuwamy kolejno e i d , a następnie generujemy las (c, e) . Usuwamy e , następnie c i generujemy kolejne listy: (d) , (d, e) , (e) . Nie ma już jednak więcej drzew rozpinających.

Drzewo rozpinające grafu - przykład



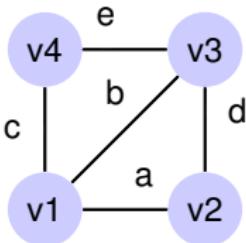
- Usuwamy kolejno krawędzie e i d , a kolejną listą jest: (b, e) . Usuwamy e , a następnie b i tworzymy listy: (c) , (c, d) , (c, d, e) . Mamy kolejne drzewo rozpinające.

Drzewo rozpinające grafu - przykład

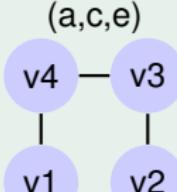
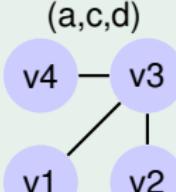
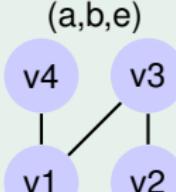
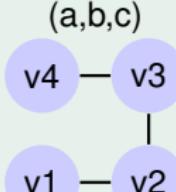
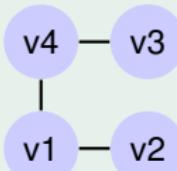
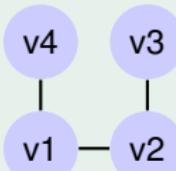
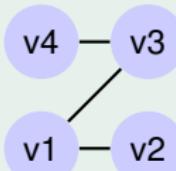
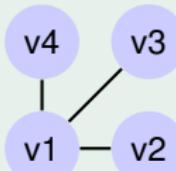


- Usuwamy kolejno krawędzie e i d , a kolejną listą jest: (b, e) . Usuwamy e , a następnie b i tworzymy listy: (c) , (c, d) , (c, d, e) . Mamy kolejne drzewo rozpinające.
- Usuwamy kolejno e i d , a następnie generujemy las (c, e) . Usuwamy e , następnie c i generujemy kolejne listy: (d) , (d, e) , (e) . Nie ma już jednak więcej drzew rozpinających.

Drzewo rozpinające grafu - przykład



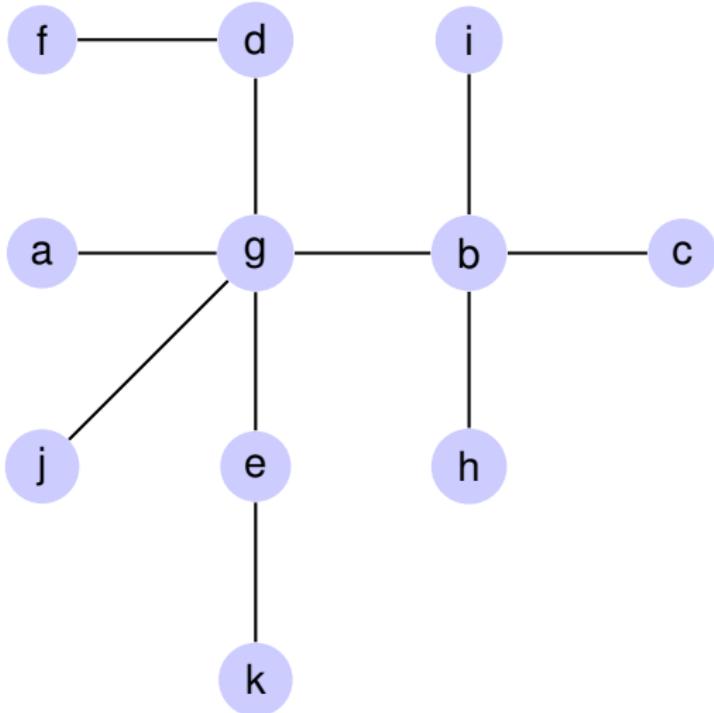
Drzewa rozpinające



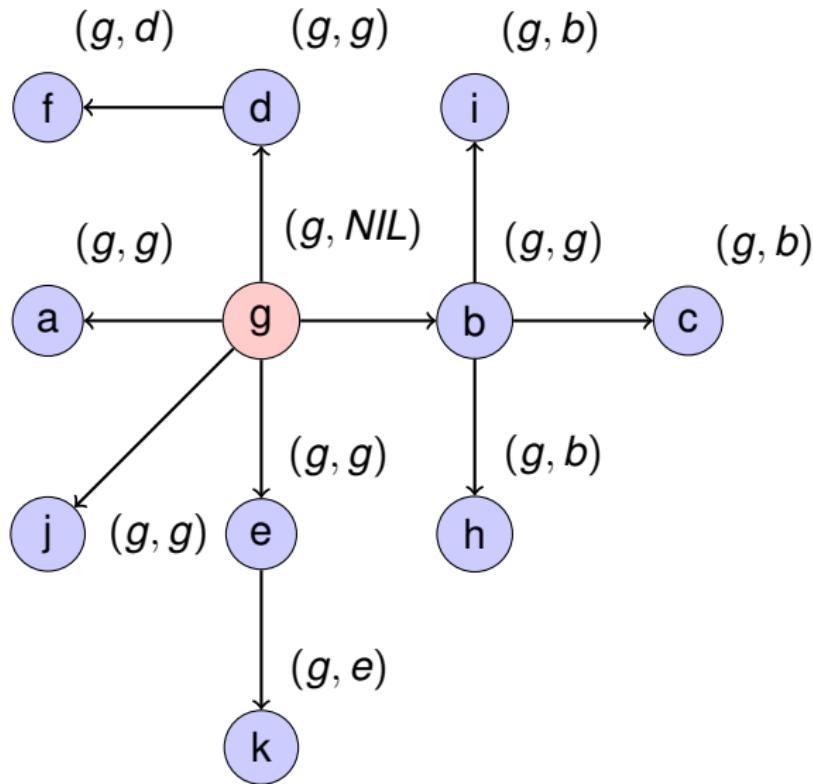
Generowanie drzew rozpinających grafu - założenia

- Należy znaleźć szybki sposób decydowania czy dodanie krawędzi nie spowoduje powstania cyklu. W tym celu w każdym drzewie wyróżnimy jeden wierzchołek - **korzeń**.
- Każdemu wierzchołkowi v w grafie przyporządkujemy dwa atrybuty:
 - $Korzen[v]$ - tzw. korzeń drzewa, w którym znajduje się wierzchołek v .
 - $Poprzednik[v]$ - poprzednik wierzchołka v na jedynej ścieżce łączącej v z korzeniem.
- Poprzednik korzenia jest nieokreślony; w algorytmach zakładamy, że jest równy 0 lub NIL .

Założenia - przykład



Założenia - przykład



Generowanie drzew rozpinających grafu - idea

Proces generowania wszystkich drzew rozpinających wymaga dwóch operacji:

- Jeżeli kolejna krawędź z listy nie zamyka cyklu (jest zaakceptowana), to dodanie jej powoduje połączenie dwóch drzew T_1 i T_2 w jedno nowe drzewo T i w związku z tym należy dokonać odpowiednich zmian etykiet wierzchołków T (**Procedura A i B**).
- Po otrzymaniu drzewa rozpinającego lub wyczerpaniu się listy krawędzi w procesie generowania drzew, wykonujemy krok “powrotu” polegający na wyrzuceniu ostatnio dodanej krawędzi. Powoduje to rozbicie pewnego drzewa T na dwa poddrzewa T_1 i T_2 (**Procedura C**).

Procedura A - pseudokod

Zamiana etykiet wierzchołków drzewa T o korzeniu r , po operacji zamiany r na nowy korzeń v .

Algorytm NowyKorzeń(v):

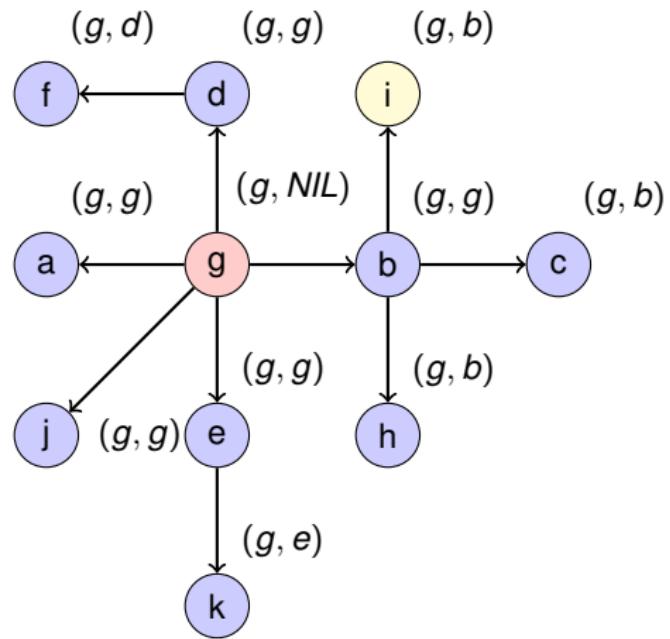
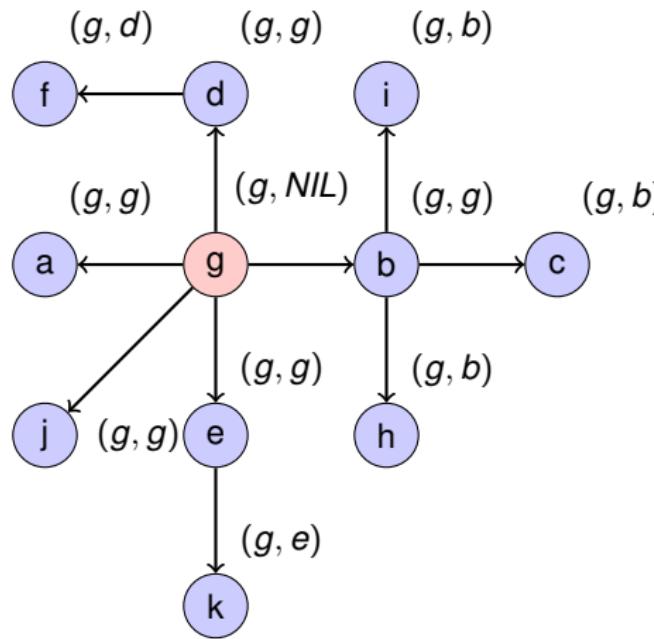
```
1: StaryKorzen := Korzen[v]
2: if StaryKorzen = v then
3:   return;
4: end if
5: v1 := 0; v2 := v;
6: repeat
7:   p := v1; v1 := v2; v2 := Poprzednik[v1]; Poprzednik[v1] := p;
8: until v1 = StaryKorzen
9: for each  $w \in V$  do
10:  if Korzen[w] = StaryKorzen then
11:    Korzen[w] = v;
12:  end if
13: end for
```

Procedura A - opis

- wierzchołkowi v jako drugą etykietę (poprzednik) przypisujemy 0 - v staje się korzeniem drzewa T .
- zmieniamy skierowanie (na przeciwe) łuków na ścieżce z r do v i odpowiednio zmieniamy drugie etykiety wierzchołków ($\neq v$) tej ścieżki (drugie etykiety pozostałych wierzchołków w drzewie T pozostają bez zmian).
- wszystkie wierzchołki drzewa T otrzymują pierwszą etykietę (określającą korzeń) równą v .

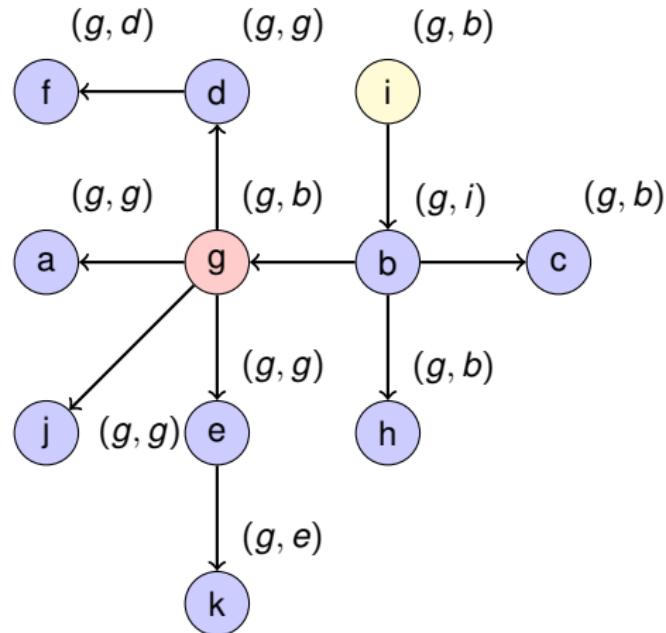
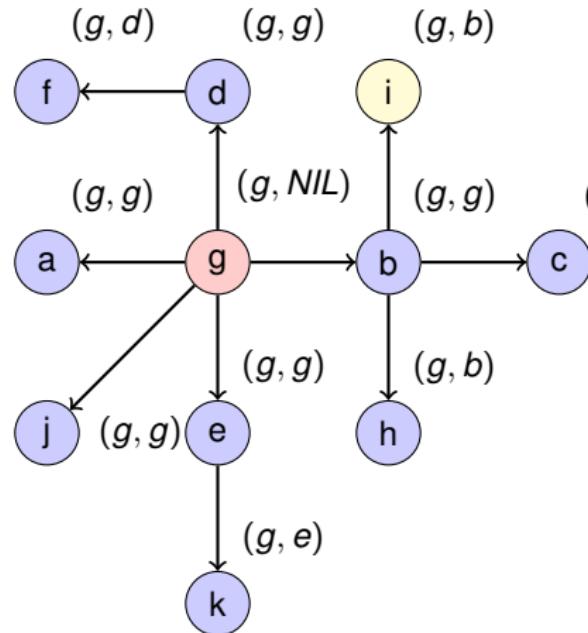
Procedura A - przykład

- operacja zamiany korzenia g na nowy korzeń i .



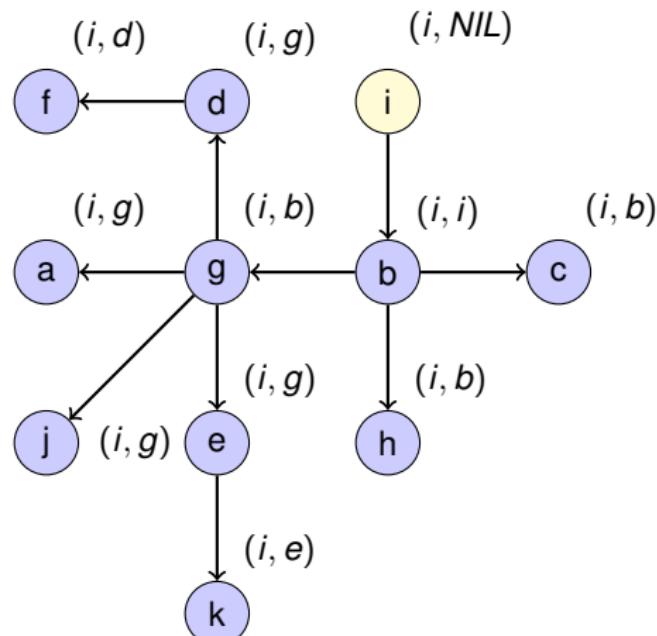
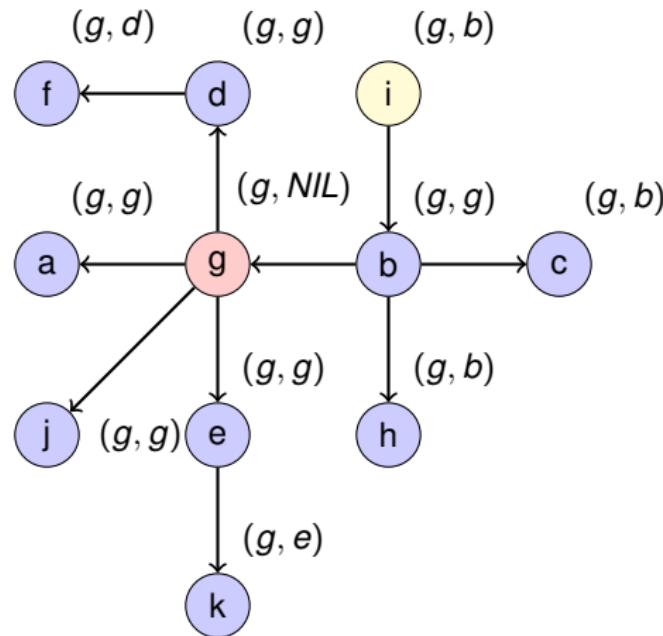
Procedura A - przykład, cd.

- zmieniamy skierowanie krawędzi na ścieżce z g do i .
- zmieniamy drugie etykiety wierzchołków ($\neq i$) tej ścieżki. Etykiety pozostałych wierzchołków w drzewie T pozostają bez zmian.



Procedura A - przykład, cd.

- Wszystkie wierzchołki drzewa T otrzymują pierwszą etykietę (określającą korzeń) równą i , a wierzchołek i dostaje drugą etykietę postaci (i, NIL) .



Procedura B - pseudokod

Zamienia etykiety wierzchołków dwóch drzew T_1 i T_2 , o korzeniach odpowiednio r_1 i r_2 , $r_1 < r_2$, po operacji ich połączenia przez dodanie krawędzi $e = (u, v)$, gdzie $u \in V(T_1)$, $v \in V(T_2)$.

Algorytm DodajKrawędź ($e = (u, v)$):

1: $v1 := Korzen[u]; v2 := Korzen[v];$

2: **if** $v2 < v1$ **then**

3: $zamien(v1, v2); zamien(u, v);$

4: **end if**

5: $NowyKorzen(v);$

6: $Poprzednik[v] := u;$

7: **for** each $w \in V$ **do**

8: **if** $Korzen[w] = v2$ **then**

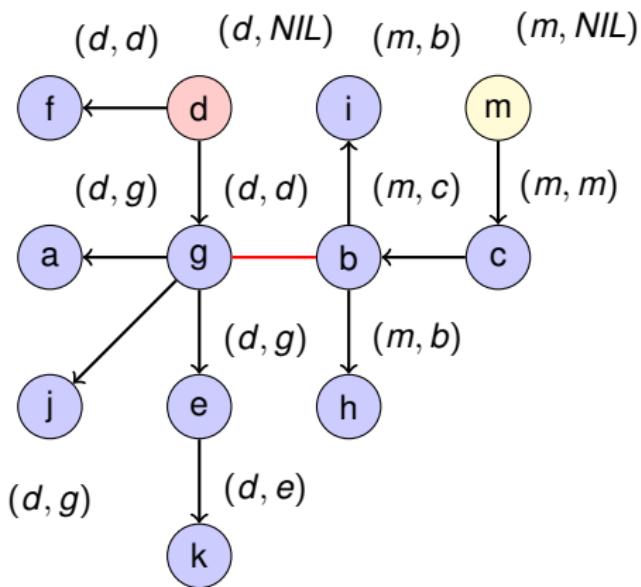
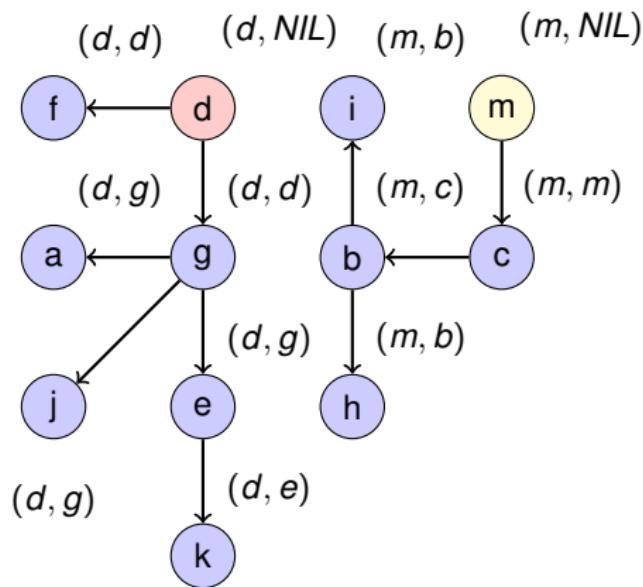
9: $Korzen[w] = v1;$

10: **end if**

11: **end for**

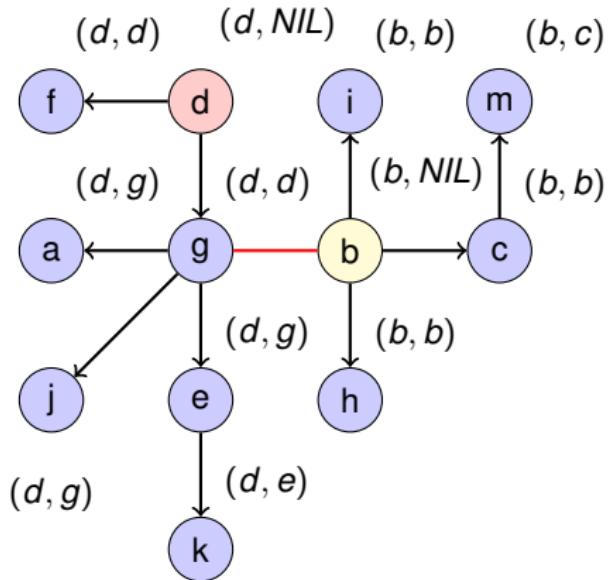
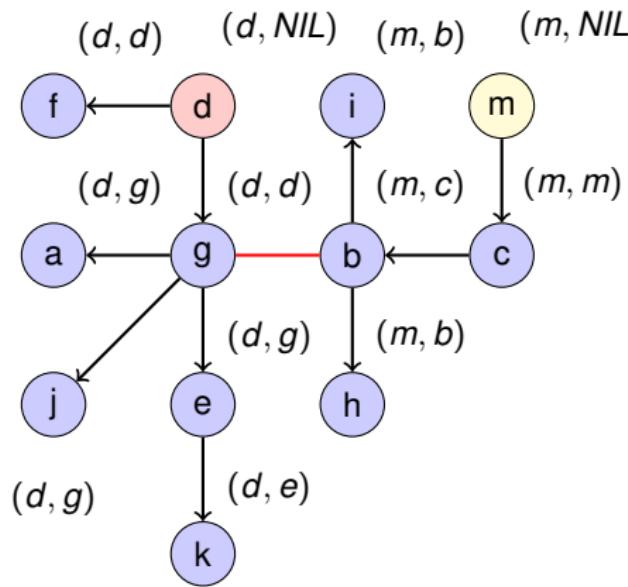
Procedura B - Przykład

- Połączenie dwóch drzew T_1 i T_2 , o korzeniach odpowiednio d i m , przez dodanie krawędzi $e = (g, b)$, gdzie $g \in V(T_1)$, $b \in V(T_2)$.



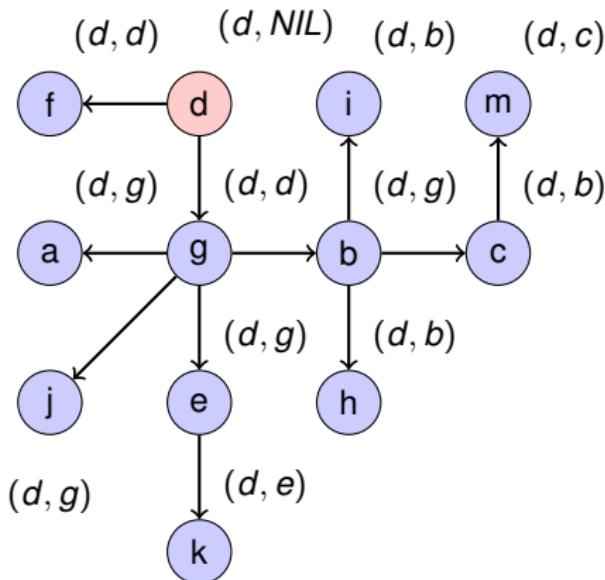
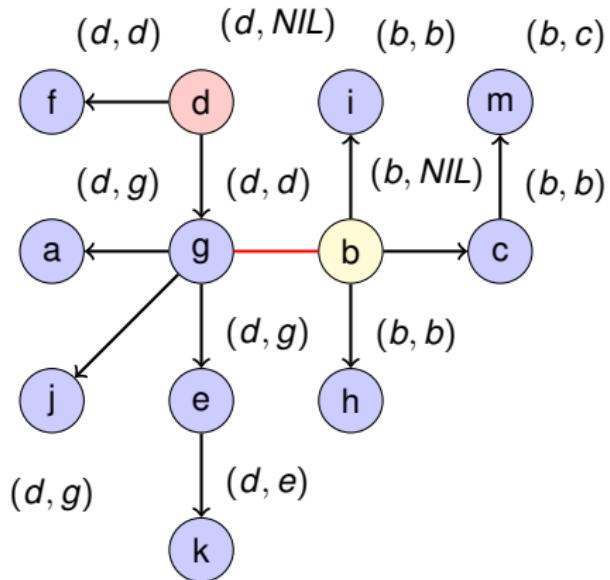
Procedura B - Przykład, cd.

- wierzchołkom w drzewie T_2 zmieniamy drugie etykiety tak jak przy zamianie korzenia z m na b (Procedura A); w drzewie T_1 drugie etykiety wierzchołków pozostają bez zmian.



Procedura B - Przykład, cd.

- wierzchołkowi b jako drugą etykietę przypisujemy g .
- wszystkie wierzchołki drzewa $T2$ otrzymują pierwszą etykietę równą d ; w drzewie $T1$ pierwsze etykiety wierzchołków pozostają bez zmian.



Procedura C - pseudokod

Zamiana etykiet wierzchołków drzewa T o korzeniu r po usunięciu z niego krawędzi $e = (u, v)$, czyli rozbiciu T na dwa drzewa T_1 i T_2 , gdzie $u \in V(T_1)$, $v \in V(T_2)$ i $r \in V(T_1)$.

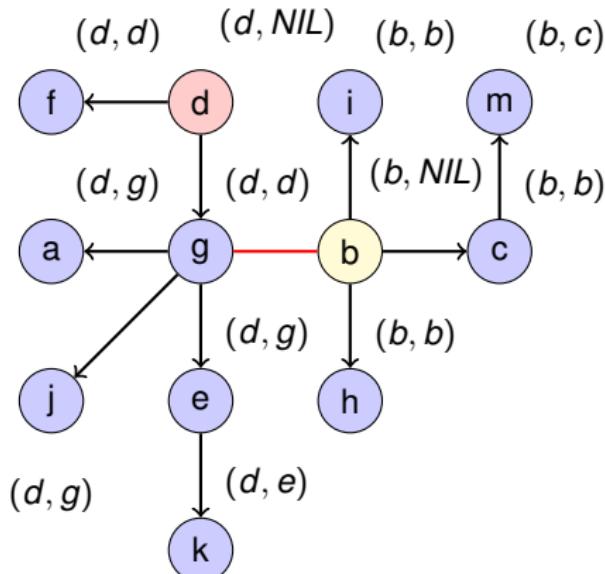
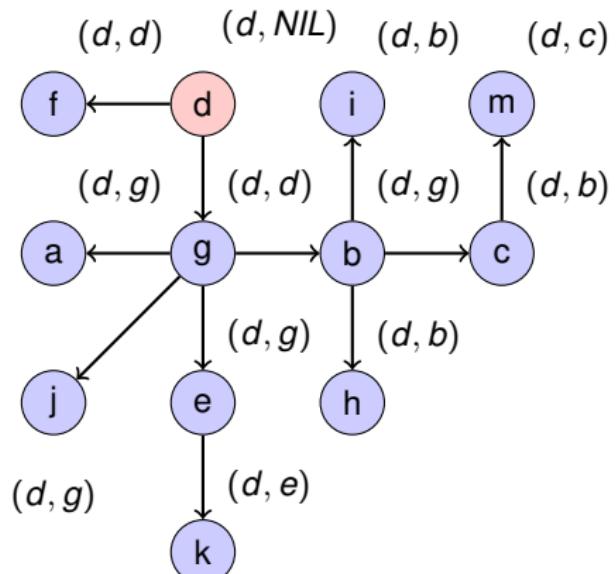
UWAGA! W naszym przypadku zawsze usuwamy ostatnio dodaną krawędź, więc wierzchołek u jest wierzchołkiem wiszącym i po usunięciu krawędzi stanie się drzewem trywialnym (jednowierzchołkowym). Zatem proces usunięcia krawędzi polega jedynie na zmianie atrybutów wierzchołka u . Realizuje to algorytm **UsuńKrawędź**.

Algorytm UsuńKrawędź($e=(u,v)$):

- 1: **if** $Poprzednik[u] = v$ **then**
- 2: $zamien(u, v)$
- 3: **end if**
- 4: $Poprzednik[v] := 0;$
- 5: $Korzen[v] = v;$

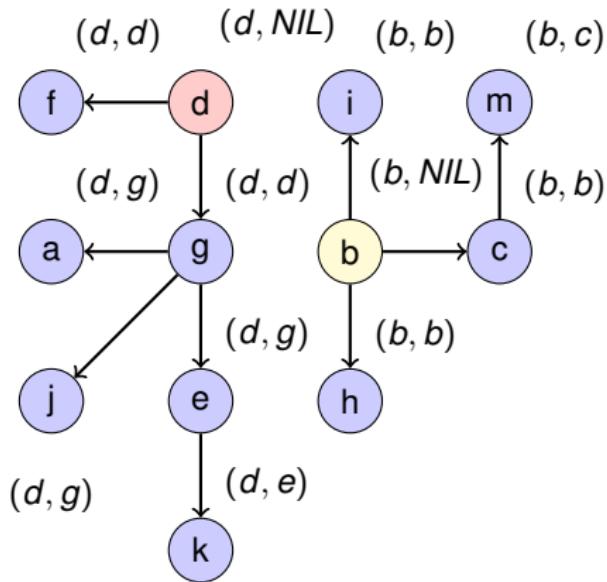
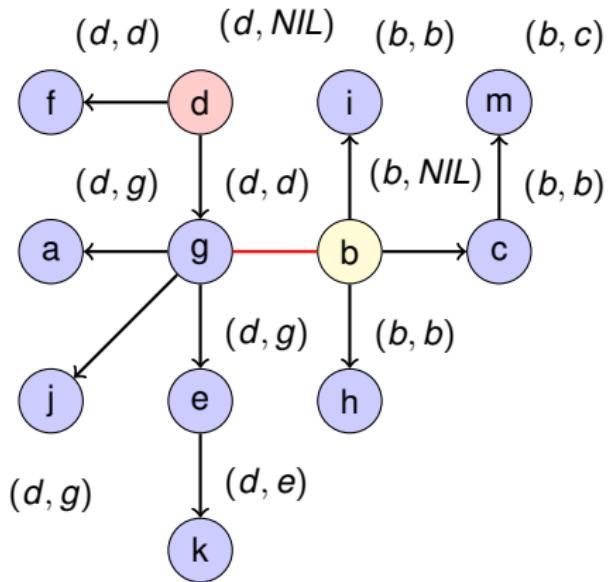
Procedura C - Przykład

- W drzewie T_1 obie etykiety wierzchołków pozostają bez zmian.
- Wierzchołkowi b przypisujemy etykietę $(b, 0)$ (staje się korzeniem drzewa T_2). Drugie etykiety pozostałych wierzchołków w drzewie T_2 pozostają bez zmian, ale otrzymują pierwszą etykietę równą b .



Procedura C - Przykład, cd.

- Usunięcie krawędzi (g, b) .



Algorytm generowania wszystkich drzew rozpinających grafu

Oznaczenia w algorytmie WSZYSTKIE DRZEWIA:

- $\nu(G)$, czyli n - oznacza liczbę wierzchołków grafu G .
- $\varepsilon(G)$, czyli m - oznacza liczbę krawędzi grafu G .
- Drzewo - oznacza wygenerowane drzewo rozpinające.
- Drzewo[i] - oznacza i -ta krawędź tego drzewa.

Algorytm generowania wszystkich drzew rozpinających grafu

1. Utwórz listę krawędzi biorąc na jej początek krawędzie incydentne z dowolnie wybranym wierzchołkiem v^* :

$$e_1, e_2, \dots, e_{d(v^*)}, e_{d(v^*)+1}, \dots, e_m$$

Dla każdego wierzchołka $v \in V(G)$:

$$\text{Korze}[v] = v \quad \text{Poprzednik}[v] = 0$$

Nadajemy wartości początkowe zmiennym:

$k = 1$ (k - licznik krawędzi z listy),

$\text{koniec} = d(v^*)$.

Algorytm generowania wszystkich drzew rozpinających grafu

2. Jeżeli $k = m + 1$, to przejdź do kroku 5. W przeciwnym razie badamy k -tą krawędź z listy: $e_k = (u_k, v_k)$.
 - Jeżeli $\text{Korze}(u_k) = \text{Korze}(v_k)$, to u_k i v_k są w tym samym drzewie i krawędź e_k zamyka cykl - odrzucamy ją; $k = k + 1$; przejdź do kroku 2;
 - Jeżeli $\text{Korze}(u_k) \neq \text{Korze}(v_k)$, to krawędź akceptuj i przejdź do kroku 3;
3. Połącz dwa drzewa krawędzią $e_k = (u_k, v_k)$ zgodnie z procedurą **DODAJKRAWĘDŹ**($e_k = (u_k, v_k)$);

Algorytm generowania wszystkich drzew rozpinających grafu

4. Jeżeli liczba zaakceptowanych krawędzi jest równa $n - 1$, to mamy drzewo rozpięte. Zapamiętaj je i przejdź do kroku 5, jeżeli krawędzi zaakceptowanych jest mniej, to $k = k + 1$ i przejdź do kroku 2;
5. Badamy ostatnio zaakceptowaną krawędź. Powiedzmy, że jest nią krawędź e_I . Jeżeli $e_I = e_{koniec}$ i nie ma już innych zaakceptowanych krawędzi, to STOP (mamy wyznaczone wszystkie rozpięte drzewa). W przeciwnym razie odrzuć krawędź e_I i zmień etykiety zgodnie z procedurą **USUŃKRAWĘDŹ** ($e_I = (u_I, v_I)$); $k = I + 1$ i przejdź do kroku 2.

Algorytm generowania wszystkich drzew rozpinających grafu

Algorithm 0.0.1: WSZYSTKIEDRZEWA($G, koniec$)

comment: Zakładamy, że $e_k = u_kv_k$

external NOWYKORZEŃ(), DODAJKRAWĘDŹ(), USUŃKRAWĘDŹ()

for each $v \in V(G)$

do $Korzeń[v] = v$; $Poprzednik[v] \leftarrow 0$

$k \leftarrow 1$; $i \leftarrow 0$

repeat

if $Korzeń[u_k] \neq Korzeń[v_k]$

then $\begin{cases} \text{DODAJKRAWĘDŹ}(e_k) \\ i \leftarrow i + 1 \\ Drzewo[i] \leftarrow e_k \end{cases}$

if $i = \nu(G) - 1$ **then output** ($Drzewo$)

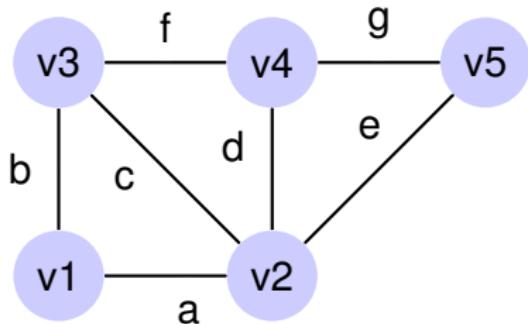
if $i = \nu(G) - 1 \vee k = \varepsilon(G)$

then $\begin{cases} \text{USUŃKRAWĘDŹ}(Drzewo[i]) \\ k \leftarrow Drzewo[i] + 1 \\ i \leftarrow i - 1 \end{cases}$

else $k \leftarrow k + 1$

until $Drzewo[1] = e_{koniec}$

Znaleźć wszystkie drzewa rozpięte grafu



- a, ab, abc, abd, abde, abdf, abdg, abef, abeg, abfg, abg, ac, acd, acde, acdf, acdg, ace, acef, aceg, acf, acfg, acg, ad, ade, adef, adeg, adf, adfg, adg, aef, aefg, aeg, af, afg, ag,
- b, bc, bcd, bcde, bcdf, bcdg, bce, bcef, bceg, bcf, bcfg, bcg, bd, bde, bdef, bdeg, bdf, bdfg, bdg, be, bef, befg, beg, bf, bfg, bg, c.

Algorytmy i Struktury Danych

Drzewa poszukiwań binarnych

dr hab. Bożena Woźna-Szcześniak
bwozna@gmail.com

Jan Długosz University, Poland

Wykład 12



Plan wykładu

- Drzewiaste struktury danych
 - Dlaczego ich potrzebujemy
 - Podstawowe definicje
 - Dynamiczne realizacje
- Drzewa poszukiwań binarnych (BST)
 - Definicja
 - Odwiedzanie wierzchołków
 - Wyszukiwanie danego elementu, wyszukiwanie maksimum i minimum
 - Wstawianie nowego elementu do drzewa
 - Usuwanie elementu z drzewa

Wstawianie i wyszukiwanie kluczy

- Tablice nieposortowane
- Tablice posortowane
- Drzewa poszukiwań binarnych

Tablice nieposortowane

- Operacja wstawiania na koniec – Złożoność $O(1)$

Tablice nieposortowane

- Operacja wstawiania na koniec – Złożoność $O(1)$
- Operacja wstawiania na początek – Złożoność $O(n)$

Tablice nieposortowane

- Operacja wstawiania na koniec – Złożoność $O(1)$
- Operacja wstawiania na początek – Złożoność $O(n)$
- Operacja wyszukiwania elementu (wyszukiwanie liniowe) – Złożoność $O(n)$

Tablice nieposortowane

- Operacja wstawiania na koniec – Złożoność $O(1)$
- Operacja wstawiania na początek – Złożoność $O(n)$
- Operacja wyszukiwania elementu (wyszukiwanie liniowe) – Złożoność $O(n)$
- A może jest coś lepszego ...

Tablice posortowane

- Operacja wstawiania – równoważne sortowaniu przez wstawianie, złożoność $O(n^2)$.

Tablice posortowane

- Operacja wstawiania – równoważne sortowaniu przez wstawianie, złożoność $O(n^2)$.
- Operacja wyszukiwania elementu (wyszukiwanie binarne) – Złożoność $O(\log(n))$.

Tablice posortowane

- Operacja wstawiania – równoważne sortowaniu przez wstawianie, złożoność $O(n^2)$.
- Operacja wyszukiwania elementu (wyszukiwanie binarne) – Złożoność $O(\log(n))$.

Algorytm bisekcji (Tab,n,x):

```
1:  $i := 0; j := n - 1;$ 
2: while ( $j - i > 1$ ) do
3:    $m := (i + j) \text{div} 2;$ 
4:   if  $\text{Tab}[m] \leq x$  then
5:      $i := m;$ 
6:   else
7:      $j := m;$ 
8:   end if
9: end while
10: if  $\text{Tab}[i] = x$  then
11:   return true;
12: else
13:   return false;
14: end if
```

Tablice posortowane

- Operacja wstawiania – równoważne sortowaniu przez wstawianie, złożoność $O(n^2)$.
- Operacja wyszukiwania elementu (wyszukiwanie binarne) – Złożoność $O(\log(n))$.
- Potrzebujemy czegoś lepszego ...

Algorytm bisekcji (Tab,n,x):

```
1:  $i := 0; j := n - 1;$ 
2: while ( $j - i > 1$ ) do
3:    $m := (i + j) \text{div} 2;$ 
4:   if  $\text{Tab}[m] \leq x$  then
5:      $i := m;$ 
6:   else
7:      $j := m;$ 
8:   end if
9: end while
10: if  $\text{Tab}[i] = x$  then
11:   return true;
12: else
13:   return false;
14: end if
```

Drzewa



Drzewa

- Drzewa poszukiwań binarnych (BST)



Drzewa

- Drzewa poszukiwań binarnych (BST)
- Kopce



Drzewa

- Drzewa poszukiwań binarnych (BST)
- Kopce
- Drzewa AVL - nazwa *AVL* pochodzi od nazwisk rosyjskich matematyków:
Gieorgij Adelson-Wielskij i Jewgienij Łandis



Drzewa

- Drzewa poszukiwań binarnych (BST)
- Kopce
- Drzewa AVL - nazwa *AVL* pochodzi od nazwisk rosyjskich matematyków:
Gieorgij Adelson-Wielskij i Jewgienij Łandis
- Drzewa Czerwono-Czarne



Drzewa

- Drzewa poszukiwań binarnych (BST)
- Kopce
- Drzewa AVL - nazwa *AVL* pochodzi od nazwisk rosyjskich matematyków:
Gieorgij Adelson-Wielskij i Jewgienij Łandis
- Drzewa Czerwono-Czarne
- B-drzewa
- ...



Definicja

Drzewem nazywamy spójny i acykliczny graf nieskierowany.

- Graf jest spójny, gdy dowolne dwa wierzchołki są połączone drogą.
- Graf jest acykliczny, jeśli nie posiada cyklu.

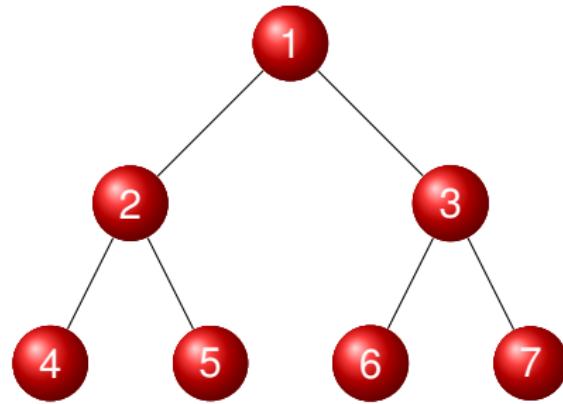
Definicja

Drzewem nazywamy **spójny i acykliczny** graf nieskierowany.

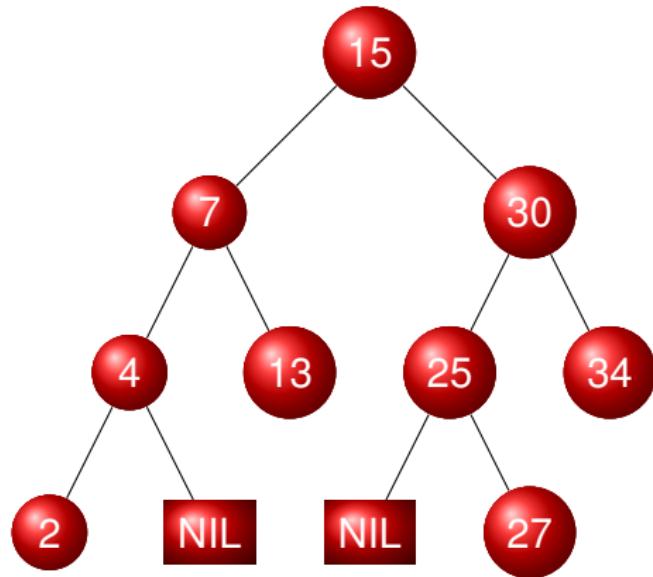
- Graf jest spójny, gdy dowolne dwa wierzchołki są połączone drogą.
 - Graf jest acykliczny, jeśli nie posiada cyklu.
-
- Drzewo, w którym wyróżniony jest jeden, charakterystyczny wierzchołek nazywamy **drzewem z korzeniem**.
 - **Korzeń** jest jedynym elementem drzewa, który nie posiada poprzednika (rodzica). Dla każdego innego wierzchołka określony jest dokładnie jeden rodzic.
 - Wierzchołki znajdujące się bezpośrednio pod danym węzłem nazywamy **synami (lub dziećmi)**.
 - Wierzchołki, które nie mają potomków nazywane są **liśćmi**.
 - Jeżeli liczba następców dla każdego wierzchołka wynosi co najwyżej dwa, to takie drzewo nazywamy **binarnym**.

- **Zupełne drzewo binarne** - Każdy węzeł, z wyjątkiem liści, ma dokładnie dwa następcy.
- **Drzewo poszukiwań binarnych (BST)** - Dla każdego węzła (nie będącego liściem) wszystkie wartości przechowywane w lewym poddrzewie są mniejsze od wartości tego węzła, natomiast wszystkie wartości przechowywane w prawym poddrzewie są większe od wartości w tym węźle.
- **Kopiec (sterta)** - Wartości przechowywane w następcach każdego węzła są mniejsze od wartości w danym węźle (tzw. *kopiec maksymalny*) lub wartości przechowywane w następcach każdego węzła są większe od wartości w danym węźle (tzw. *kopiec minimalny*). Drzewo jest szczerbnie wypełniane (zrównoważone) od lewego poddrzewa.

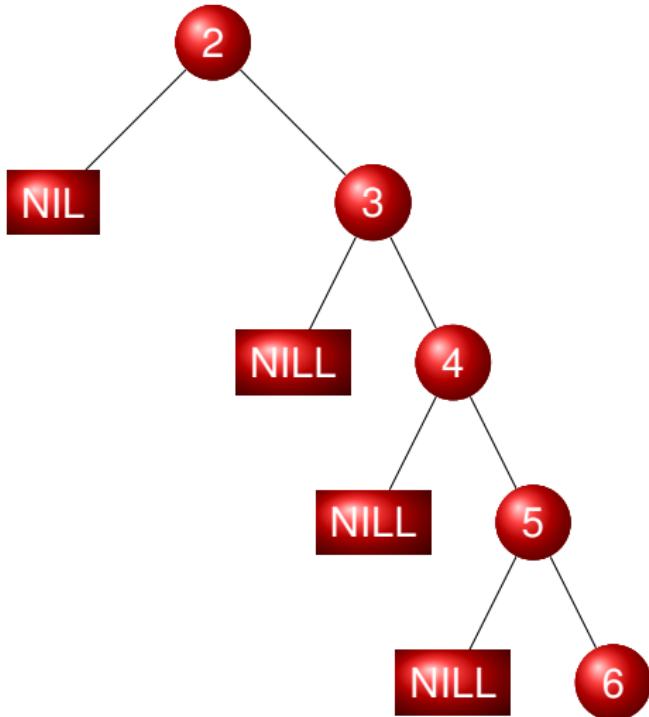
Zupełne drzewo binarne



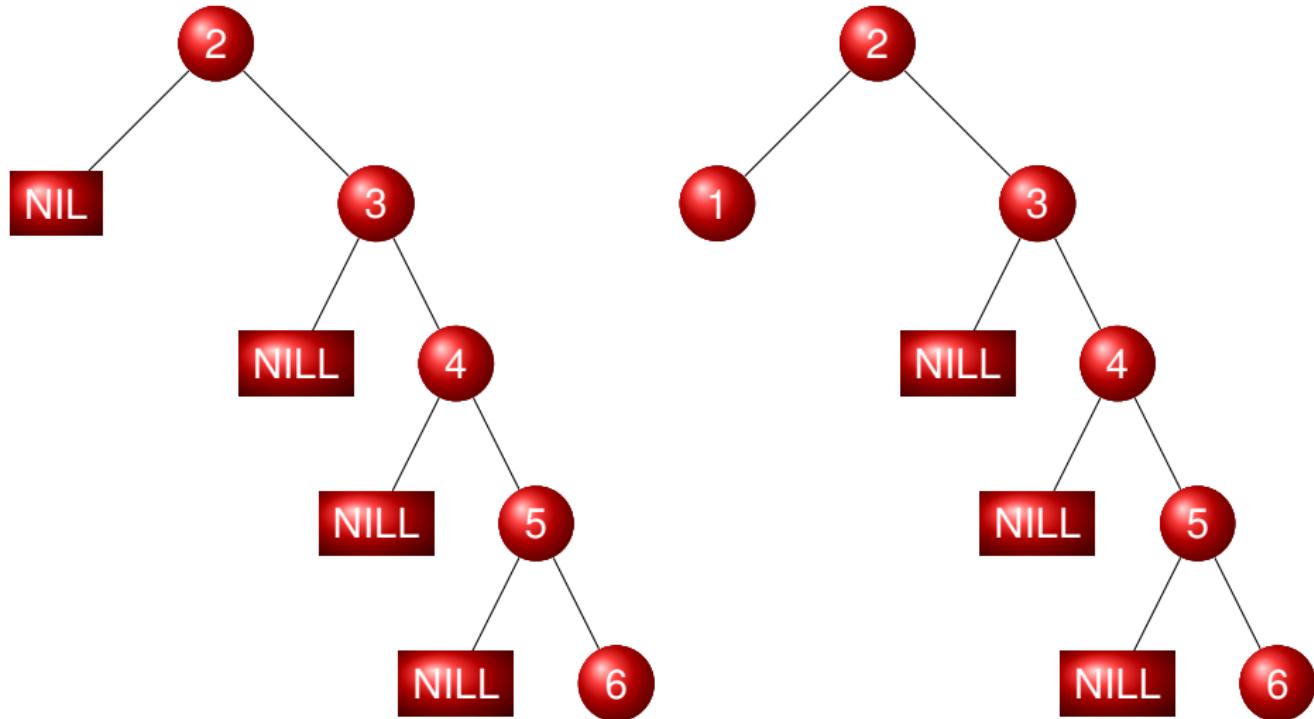
Drzewo poszukiwań binarnych



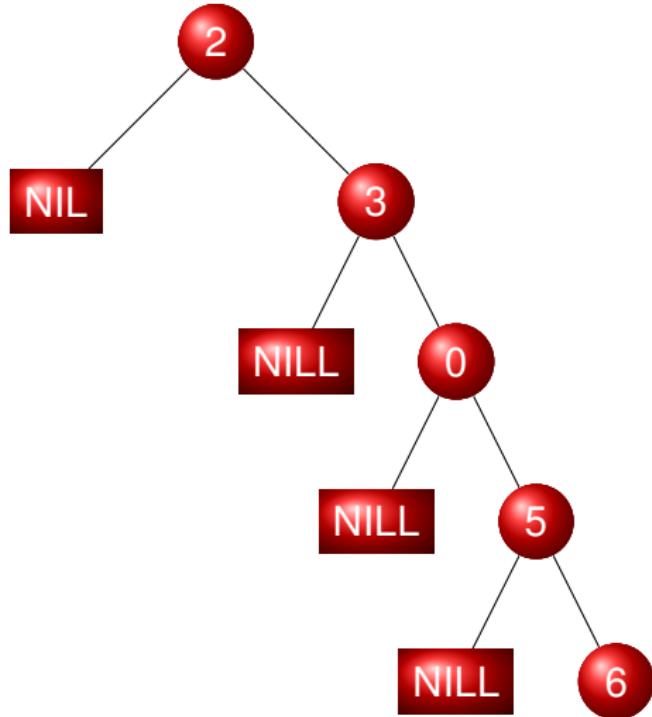
Drzewo poszukiwań binarnych



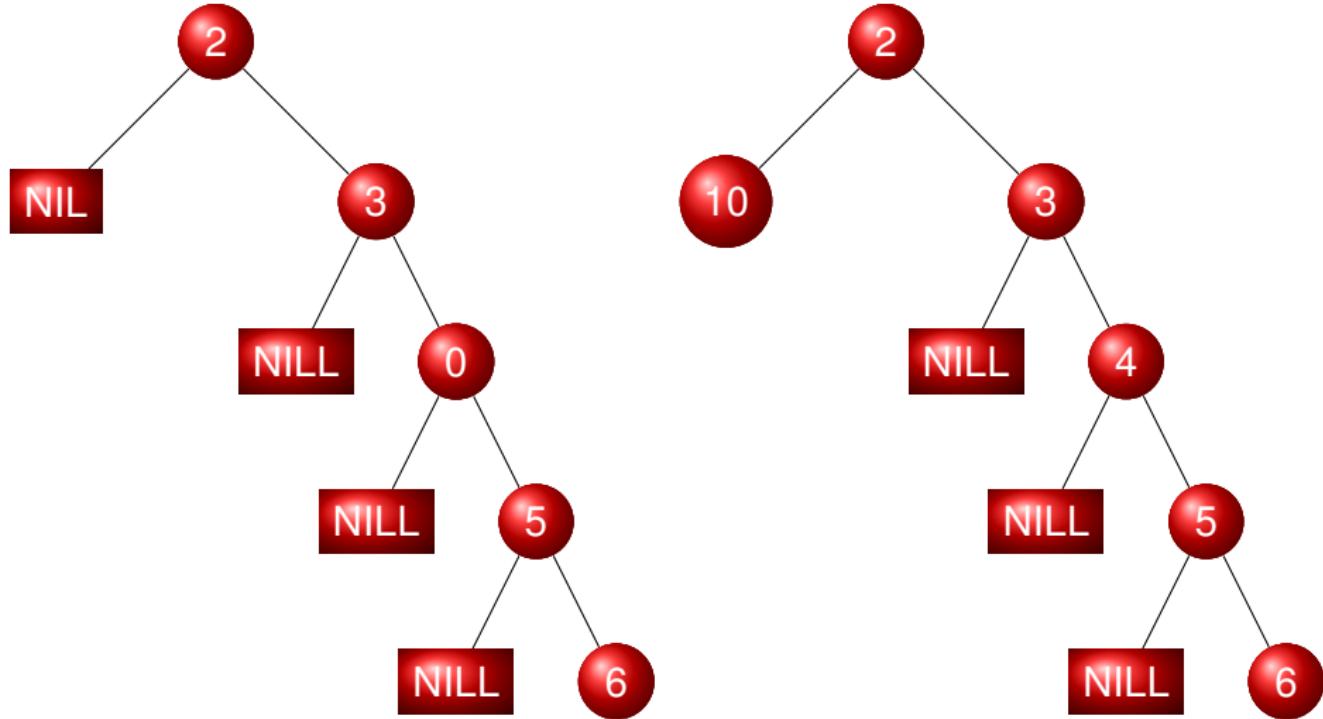
Drzewo poszukiwań binarnych



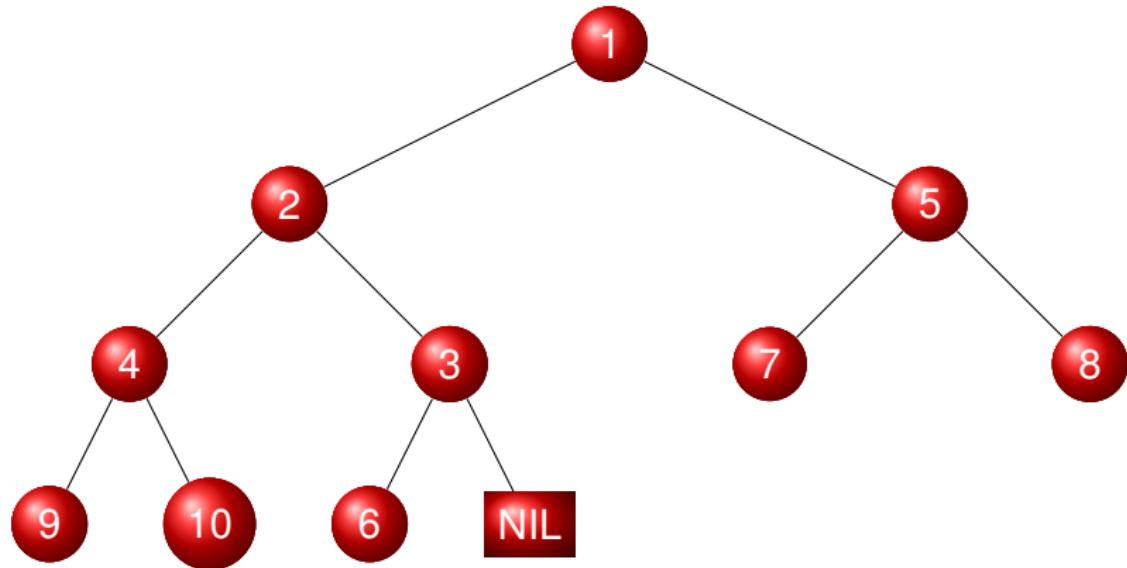
Drzewo binarne, ale nie poszukiwań binarnych



Drzewo binarne, ale nie poszukiwań binarnych



Kopiec (sterta)

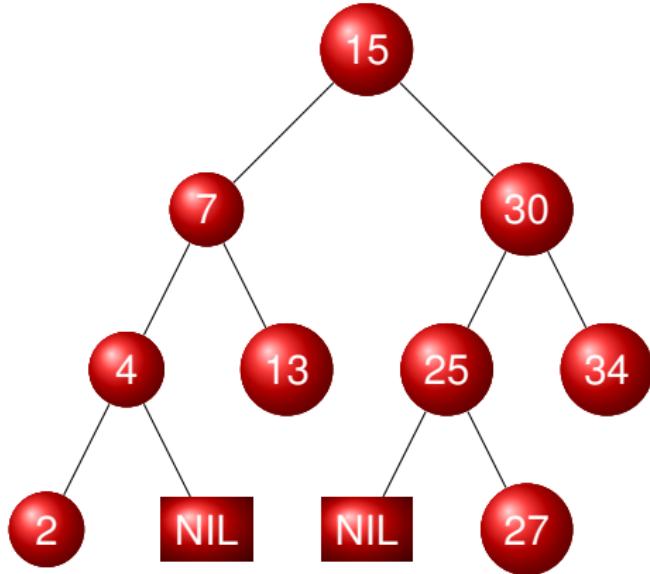


Dla każdego drzewa można określić:

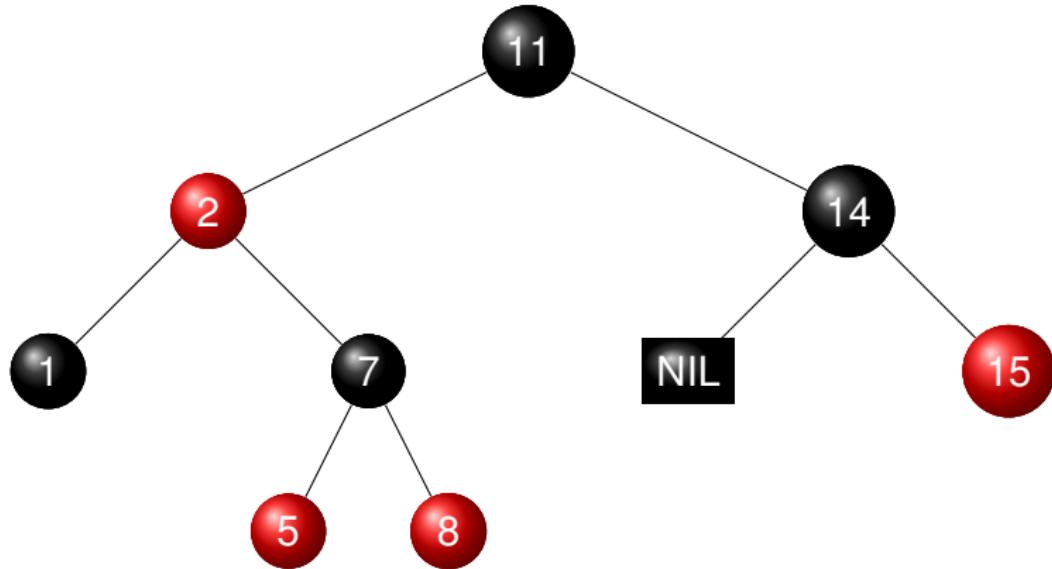
- **głębokość wierzchołka u** - liczba wierzchołków, przez które należy przejść od korzenia do wierzchołka u .
- **wysokość u** - maksymalna liczba wierzchołków na drodze od u do pewnego liścia.
- **wysokość drzewa = głębokość = wysokość korzenia + 1**
- **ścieżka z u do v** - zbiór wierzchołków, przez które należy przejść z wierzchołka u do v .
- **droga = ścieżka skierowana**.
- **stopień wierzchołka** - liczba jego bezpośrednich następców.
- **Stopień drzewa** - maksymalny stopień wierzchołka.

- **Drzewo AVL** (1962 - Adelson-Velskii, Landis) - Drzewo BST jest drzewem AVL wtedy, kiedy dla każdego wierzchołka wysokości dwóch jego poddrzew różnią się o co najwyżej jeden poziom;
- **Drzewo Czerwono-Czarne** -
 - Są to drzewa poszukiwań binarnych
 - Każdy węzeł jest czerwony lub czarny
 - Korzeń jest czarny
 - Czerwony węzeł ma zawsze czarnego ojca
 - Ilość czarnych węzłów na dowolnej ścieżce od korzenia do liścia jest taka sama

Drzewo AVL



Drzewo Czerwono-Czarne



Podstawowe operacje na strukturach drzewiastych

- Przechodzenie po drzewie - metoda wszerz (BFS), metoda w głąb (DFS)
- Wyszukanie elementu w drzewie
- Dodawanie nowego elementu do drzewa
- Usunięcie wskazanego elementu z drzewa

Algorytm przehodzenia po drzewie binarnym

- **Cel:**

- jednokrotne “odwiedzenie” każdego elementu drzewa;
- linearyzacja drzewa;

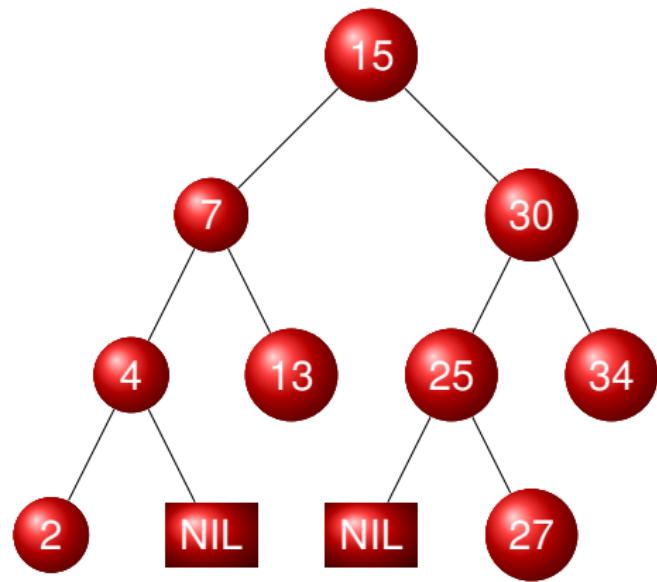
- **Dane wejściowe:** dowiązanie do korzenia drzewa “Root”;

- **Uwagi:**

- kolejność przejścia dowolna - liczba możliwych ścieżek w drzewie o n węzłach wynosi $n!$ (permutacja);
- najczęściej stosowane sposoby przeglądania: *wszerz* i *w głąb*;

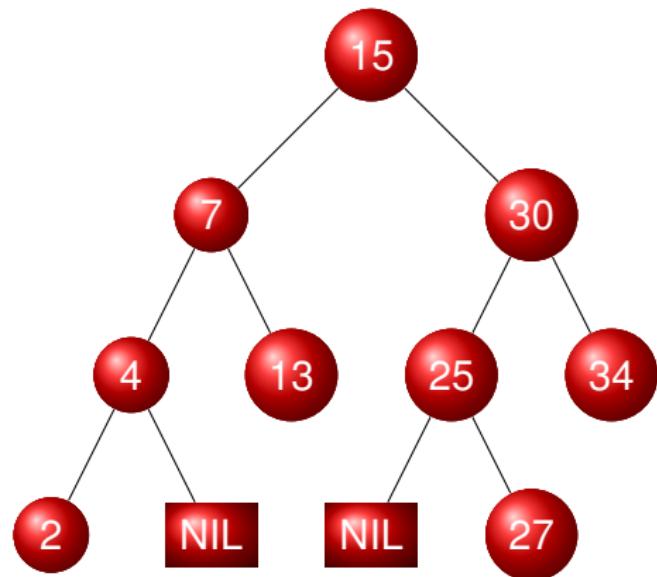
Metody przehodzenia po drzewie binarnym: BFS (wszerz)

- Przechodzenie wszerz polega na odwiedzaniu kolejno każdego węzła od najwyższego (najniższego) poziomu i przehodzeniu kolejno po tych poziomach od góry w dół (od dołu do góry) i od lewej do prawej lub od prawej do lewej.



Metody przehodzenia po drzewie binarnym: BFS (wszerz)

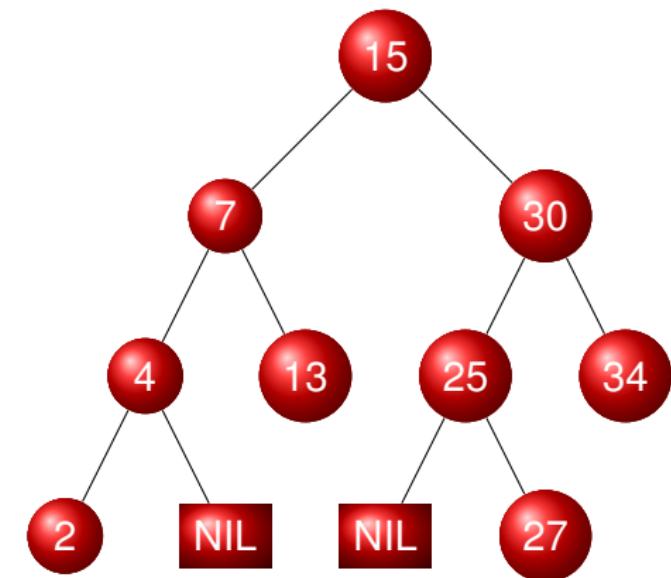
- Przechodzenie wszerz polega na odwiedzaniu kolejno każdego węzła od najwyższego (najniższego) poziomu i przehodzeniu kolejno po tych poziomach od góry w dół (od dołu do góry) i od lewej do prawej lub od prawej do lewej.
- Wynik: **15, 7, 30, 4, 13, 25, 34, 2, 27.**



Metody przehodzenia po drzewie binarnym: BFS (wszerz)

- Przechodzenie wszerz polega na odwiedzaniu kolejno każdego węzła od najwyższego (najniższego) poziomu i przehodzeniu kolejno po tych poziomach od góry w dół (od dołu do góry) i od lewej do prawej lub od prawej do lewej.

- Wynik: **15, 7, 30, 4, 13, 25, 34, 2, 27.**
- Wynik: **27, 2, 34, 25, 13, 4, 30, 7, 15.**



BFS - algorytm

Niech węzeł pewnego drzewa binarnego będzie postaci:

```
struct BSTNode
{
    BSTNode * left; // dowiezanie do lewego potomka
    BSTNode * right; //dowiezanie do prawgo potomka
    int data; // dane elementarne
};
```

Algorytm BFS(root):

- 1: *enqueueq(q, root);* //q jest kolejką
- 2: **while** *not empty(q)* **do**
- 3: *v := first(q); print(v.data); dequeueq(q);*
- 4: **if** *not empty(v.left)* **then**
- 5: *enqueueq(q, v.left);*
- 6: **end if**
- 7: **if** *not empty(v.right)* **then**
- 8: *enqueueq(q, v.right);*
- 9: **end if**
- 10: **end while**

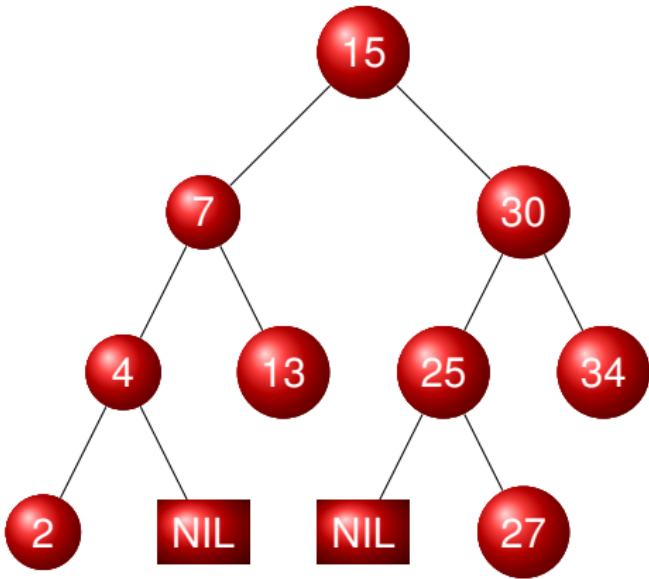
BFS - implementacja w C++

```
void BST::broadwise(BSTNode* localRoot)
{
    if (localRoot == nullptr)    return;
    Queue <BSTNode*>q;
    q.inject(localRoot);
    while (!q.empty()) {
        BSTNode* node = q.front();
        q.eject();
        if (node->left != nullptr) {
            q.inject(node->left);
        }
        if (node->right != nullptr) {
            q.inject(node->right);
        }
        cout << node->data << " ";
    }
}
```

Metody przechodzenia po drzewie binarnym: DFS (w głąb)

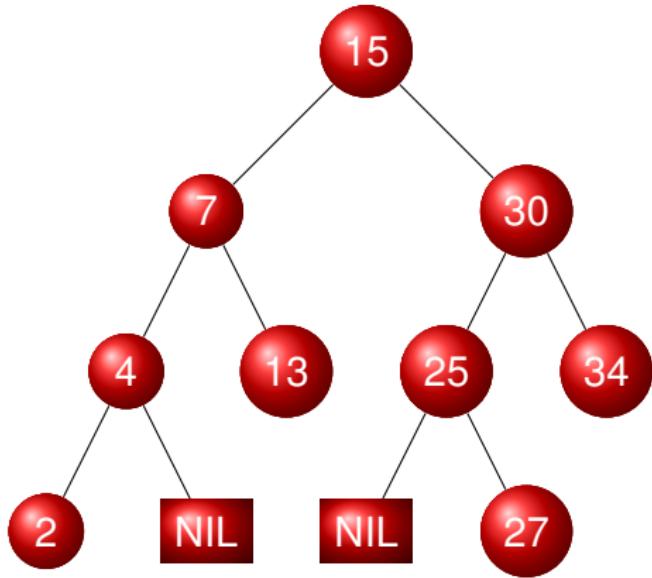
- Wersja “inorder” - LVR (porządek symetryczny)
 - Przejście do lewego poddrzewa (L);
 - Odwiedzenie węzła (V);
 - Przejście do prawego poddrzewa (R);
- Wersja “preorder” - VLR (porządek prosty)
 - Odwiedzenie węzła (V);
 - Przejście do lewego poddrzewa (L);
 - Przejście do prawego poddrzewa (R);
- Wersja “postorder” - LRV (porządek odwrotny)
 - Przejście do lewego poddrzewa (L);
 - Przejście do prawego poddrzewa (R);
 - Odwiedzenie węzła (V);

DFS - przykład



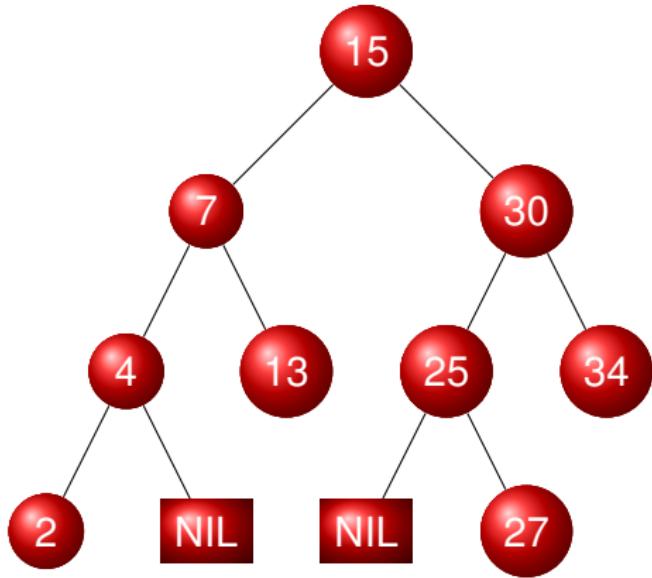
DFS - przykład

- Inorder: 2, 4, 7, 13, 15, 25, 27, 30, 34 .



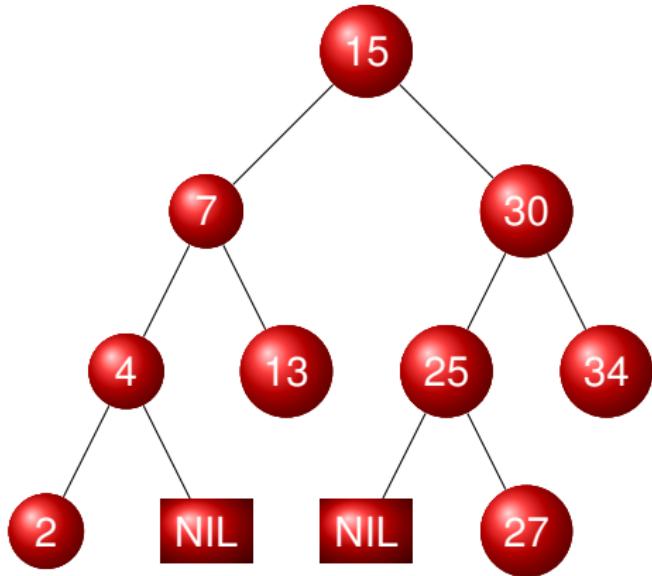
DFS - przykład

- Inorder: 2, 4, 7, 13, 15, 25, 27, 30, 34 .
- Preorder: 15, 7, 4, 2, 13, 30, 25, 27, 34.

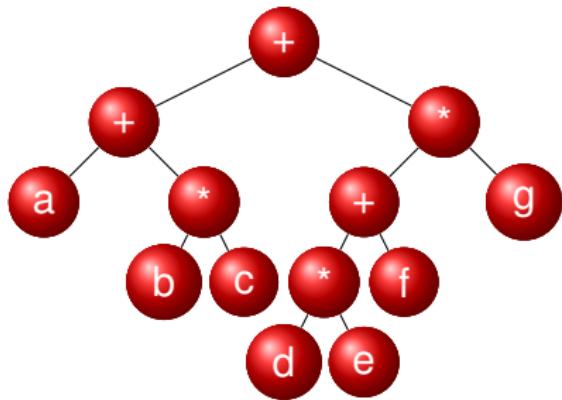


DFS - przykład

- Inorder: 2, 4, 7, 13, 15, 25, 27, 30, 34 .
- Preorder: 15, 7, 4, 2, 13, 30, 25, 27, 34.
- Postorder: 2, 4, 13, 7, 27, 25, 34, 30, 15.

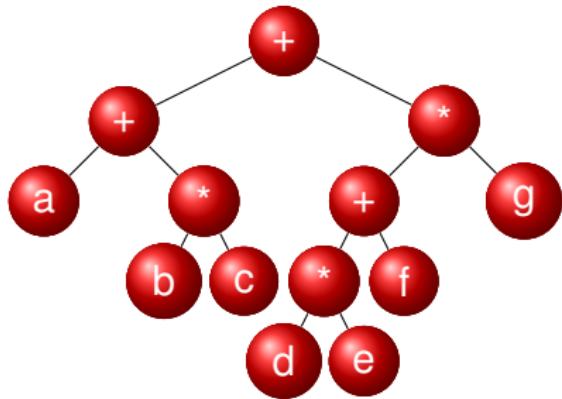


DFS - przykład



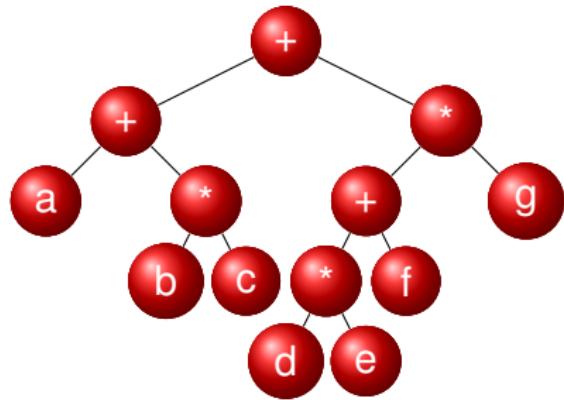
DFS - przykład

- Inorder: $a+b*c + (d*e+f)*g$.



DFS - przykład

- Inorder: $a+b*c + (d*e+f)*g$.
- Postorder: $abc^*+de^*f+g^*$.



```
void BST::preorder(const BSTNode * x);
```

```
void BST::preorder(const BSTNode * x)
{
    if( x!= nullptr)
    {
        cout << x->data << endl;
        preorder(x->left);
        preorder(x->right);
    }
}
```

```
void BST::inorder(const BSTNode * x);
```

```
void BST::inorder(const BSTNode * x)
{
    if(x != nullptr)
    {
        inorder(x->left);
        cout << x->data << endl;
        inorder(x->right);
    }
}
```

```
void BST::postorder(const BSTNode * x)
```

```
void BST::postorder(const BSTNode * x)
{
    if(x!= nullptr)
    {
        postorder(x->left);
        postorder(x->right);
        cout << x->data << endl;
    }
}
```

Nierekurencyjny DFS - algorytm preOrder

Algorytm preOrder(root):

```
1: push(q, root); //q jest stosem
2: while not empty(q) do
3:   v := top(q); print(v.data); pop(q);
4:   if not empty(v.left) then
5:     push(q, v.left);
6:   end if
7:   if not empty(v.right) then
8:     push(q, v.right);
9:   end if
10: end while
```

Nierekurencyjny DFS - algorytm inOrder, implementacja w C++

Niech węzeł pewnego drzewa binarnego będzie postaci:

```
struct BSTNode
{
    BSTNode * left, * right; // dowiązanie do lewego i prawego potomka
    int data; // dane elementarne
    bool visit; // informacja o odwiedzinach
};

void BST::IterativeInOrder(BSTNode * localRoot)
{
    STACK<BSTNode*> globalStack;
    while (true) {
        while (localRoot != nullptr) {
            globalStack.push(localRoot);
            localRoot = localRoot->left;
        }
        if (globalStack.empty()) return;
        localRoot = globalStack.top();
        cout << localRoot->data << " ";
        globalStack.pop();
        localRoot = localRoot->right;
    }
}
```

Nierekurencyjny DFS - algorytm postOrder, implementacja w C++

```
void BST::IterativePostOrder(BSTNode* localRoot)
{
    STACK<BSTNode*> globalStack;
    BSTNode *v;
    while(! (globalStack.empty()) || (localRoot!=nullptr)) {
        while(localRoot!=nullptr) {
            v = localRoot;
            v->visit = false;
            globalStack.push(v);
            localRoot = localRoot->left;
        }
        v = globalStack.top();
        globalStack.pop();
        if (! (v->visit)) {
            v->visit = true;
            globalStack.push(v);
            localRoot = v->right;
        } else {
            cout << v->data << " ";
            localRoot=nullptr;
        }
    }
}
```

Operacja wyszukiwania elementu w BST

- **Cel:**

- uzyskanie dowiązania do węzła;
- można je interpretować jako identyfikację węzła;

- **Dane wejściowe:**

- dowiązanie do korzenia drzewa “Root”;
- Kryterium poszukiwania, np. wartość danej elementarnej;

- **Uwagi:**

- kolejność przeszukiwania dowolna - w skrajnym przypadku należy przejrzeć wszystkie węzły w drzewie (złożoność $O(n)$);
- stosowane rozwiązania: pętla lub rekurencja;

Algorytm wyszukiwania elementu w BST

Niech węzeł pewnego drzewa binarnego będzie postaci:

```
struct BSTNode  
{  
    BSTNode * left, * right;  
    int data;  
    bool visit;  
};
```

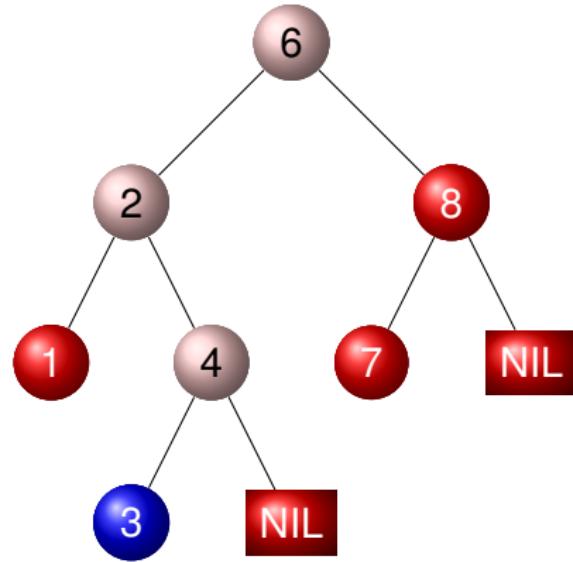
Algorytm find(root, value):

```
1: while node! = NULL do  
2:     if Value == node.data then  
3:         return node;  
4:     else if Value < node.data then  
5:         node = node.left;  
6:     else if Value > node.data then  
7:         node = node.right;  
8:     end if  
9: end while  
10: return NULL;
```

Algorytm wyszukiwania elementu w BST - przykład poszukiwanie klucza 3

- Korzeń 6, Klucz < 6, Idź na lewo,
- Wierzchołek 2, Klucz > 2, Idź na prawo,
- Wierzchołek 4, Klucz < 4, Idź na lewo,
- Wierzchołek 3, Klucz == 3. Stop

Czasy operacji: Porównanie: $O(1)$
Poszukiwanie klucza: $O(\text{głębokość})$,
jeśli klucz jest $O(\text{wysokość drzewa})$,
jeśli klucza nie ma
Złożoność czasowa: $O(\text{wysokość drzewa})$

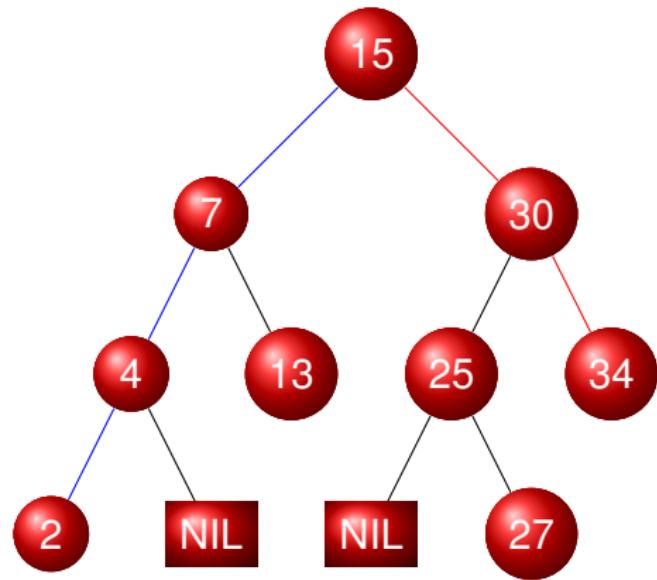


BSTNode* BST::find(int key)

```
BSTNode* BST::find(int key)
{
    if (this->root == nullptr) {
        return nullptr;
    }
    BSTNode* curr = this->root;
    while(curr->data != key) {
        if (key < curr->data) {
            curr = curr->left;
        } else {
            curr = curr->right;
        }
        if (curr == nullptr) {
            return nullptr;
        }
    }
    return curr;
}
```

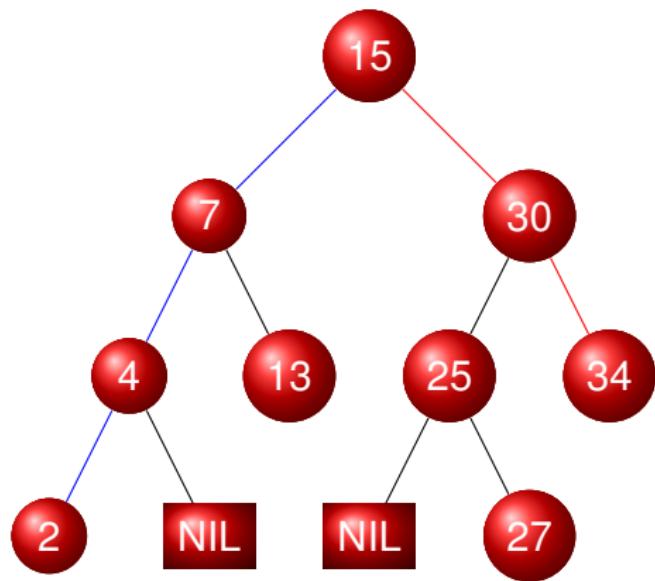
BST - wyszukiwanie elementu maksymalnego i minimalnego

- Jeżeli chcemy znaleźć element minimalny (maksymalny) w drzewie to poruszamy się maksymalnie w lewo (prawo).



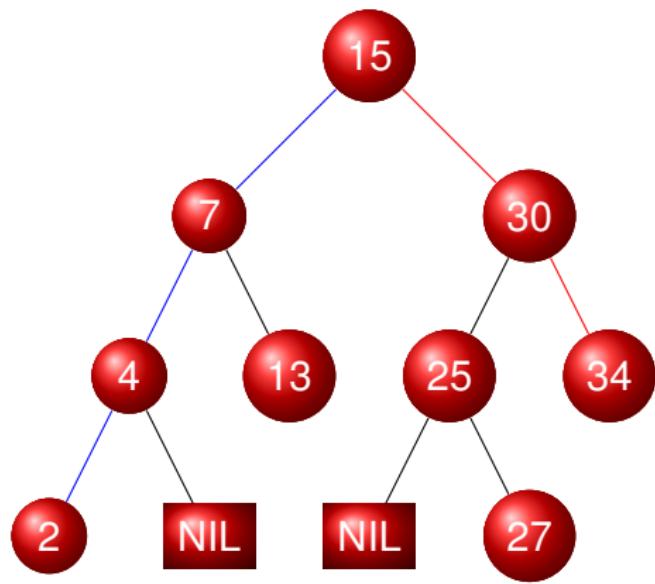
BST - wyszukiwanie elementu maksymalnego i minimalnego

- Jeżeli chcemy znaleźć element minimalny (maksymalny) w drzewie to poruszamy się maksymalnie w lewo (prawo).
- Na rysunku obok niebieska droga to poszukiwanie minimum, czerwona - maksimum.



BST - wyszukiwanie elementu maksymalnego i minimalnego

- Jeżeli chcemy znaleźć element minimalny (maksymalny) w drzewie to poruszamy się maksymalnie w lewo (prawo).
- Na rysunku obok niebieska droga to poszukiwanie minimum, czerwona - maksimum.
- Specyficzne rozmieszczenie elementów w drzewie sprawia, że element minimalny znajduje się zawsze w najbardziej “wysuniętym na lewo” węźle, a element maksymalny w najbardziej “wysuniętym na prawo” węźle drzewa.



Wstawienie nowego elementu do BST

- **Cel:** dodanie nowego elementu do drzewa;
- **Dane wejściowe:**
 - Dowiązanie do korzenia drzewa ‘Root’;
 - Nowe dane elementarne;

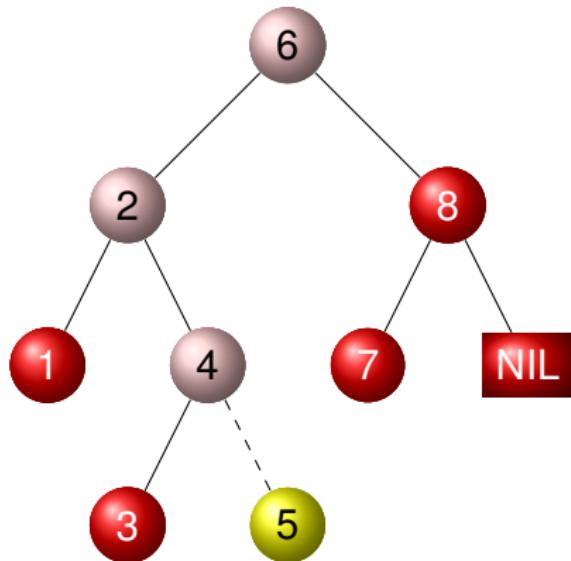
Wstawienie nowego elementu do BST - idea

- Utwórz element i ustal dane elementarne;
- Znajdź miejsce wstawienia elementu w drzewie:
 - Aby znaleźć miejsce na nowy element w drzewie BST, to począwszy od korzenia należy porównywać nowy element z węzłem i jeżeli jest on mniejszy od wartości przechowywanej w tym węźle to poruszać się w lewo po drzewie, w przeciwnym wypadku poruszać się w prawo.
 - Wędrujemy tak długo, aż dojdziemy do miejsca, w którym napotkany wskaźnik do potomka w węźle będzie wskazywał na NULL.
- Wstaw nowy węzeł w wskazane miejsce, a wspomniany wyżej wskaźnik ustaw tak, aby wskazywał na nowy węzeł.

Wstawienie nowego elementu do BST - idea

- Wierzchołek 6, Klucz < 6, Idź na lewo,
- Wierzchołek 2, Klucz > 2, Idź na prawo,
- Wierzchołek 4, Klucz > 4, Idź prawo,
- NULL, Wstaw 5.

Złożoność czasowa: **O(wysokość drzewa)**



void BST::insert(int data)

```
void BST::insert(int data)
{
    BSTNode* newNode = new BSTNode;
    newNode->data = data;
    newNode->visit = false;
    newNode->left = newNode->right = nullptr;
    count++;
    if (this->root == nullptr) {
        this->root = newNode;
    } else {
        BSTNode* curr = this->root; // zaczynamy poszukiwania od korzenia
        BSTNode* parent;
        while(true) {
            parent = curr;
            if (data < curr->data) {
                curr = curr->left;
                if (curr == nullptr) {
                    parent->left = newNode;
                    return;
                }
            } else {
                curr = curr->right;
                if (curr == nullptr) {
                    parent->right = newNode;
                    return;
                }
            }
        } // while
    }
}
```

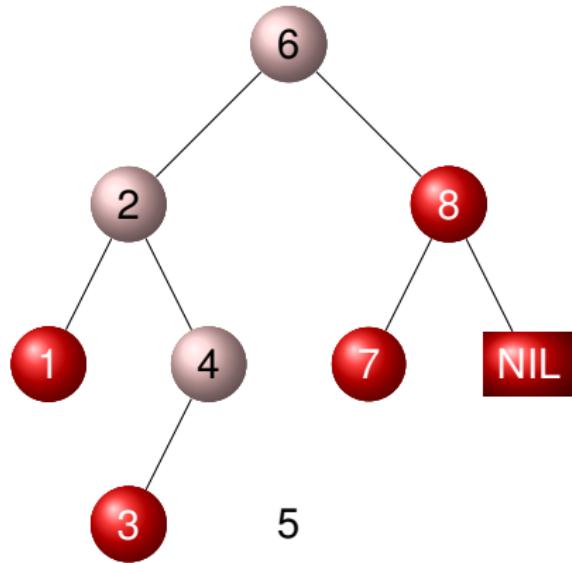
Usuwanie danego elementu do BST - idea

- **Cel:** Usunięcie węzła z drzewa;
- **Dane wejściowe:**
 - Dowiązanie do korzenia drzewa 'Root';
 - Opis elementu usuwanego, np. wartość danej elementarnej;
- **Uwagi:**
 - Przypadek 1: węzeł jest liściem;
 - Przypadek 2: węzeł ma jednego potomka;
 - Przypadek 3: węzeł ma dwóch potomków;

Przypadek 1: węzeł jest liściem;

- Usuwany węzeł nie ma potomstwa, np. węzeł 5.
- Jest to najprostsza sytuacja.
Należy usunąć ten element
(zwolnić pamięć) i zadbać o to,
aby jego rodzic wskazywał na
NULL.

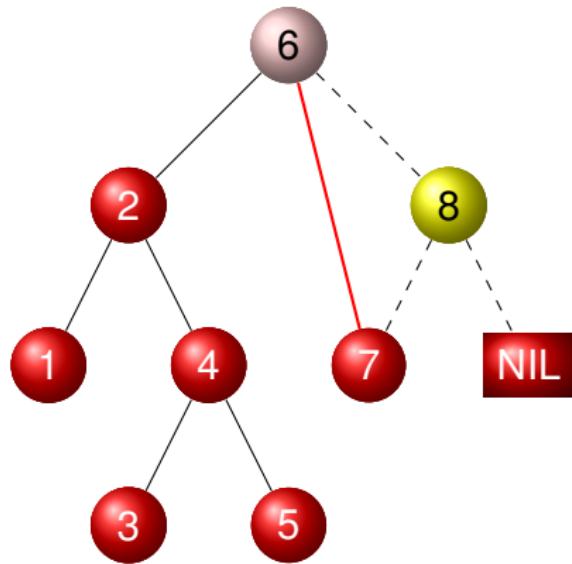
Złożoność czasowa:
O(wysokość drzewa)



Przypadek 2: węzeł ma jednego potomka;

- Usuwany węzeł posiada jednego potomka, np. 8
- Należy tutaj zadbać (oprócz zwolnienia pamięci) o to, aby rodzic usuwanego elementu wskazywał teraz zamiast na usuwany element, na jego potomka .

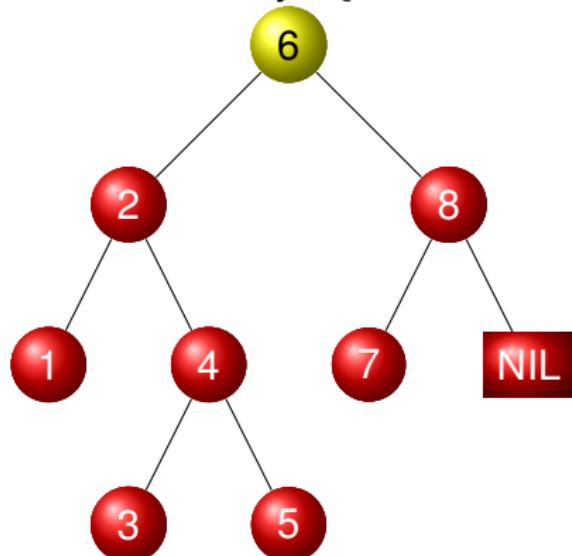
Złożoność czasowa:
O(wysokość drzewa)



Przypadek 3: węzeł ma dwóch potomków;

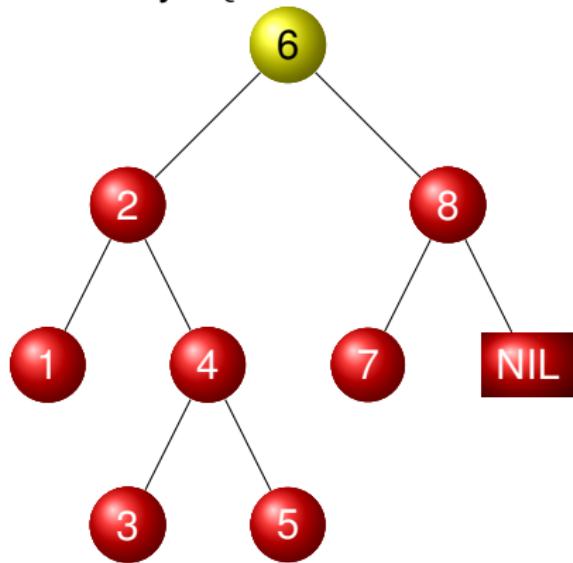
- Aby usunąć taki węzeł posiadający dwóch potomków, należy zamienić wartość z tego węzła z wartością minimalną w prawym poddrzewie usuwanego węzła lub z wartością maksymalną w lewym poddrzewie. Następnie, usuwamy element minimalny w prawym poddrzewie, ewentualnie maksymalny w lewym poddrzewie.
- Po takiej zamianie element minimalny (maksymalny) nie będzie miał potomstwa lub co najwyżej będzie miał jedynie prawego (lewego) syna.

Złożoność czasowa:
O(wysokość drzewa)
Usuwany węzeł: 6



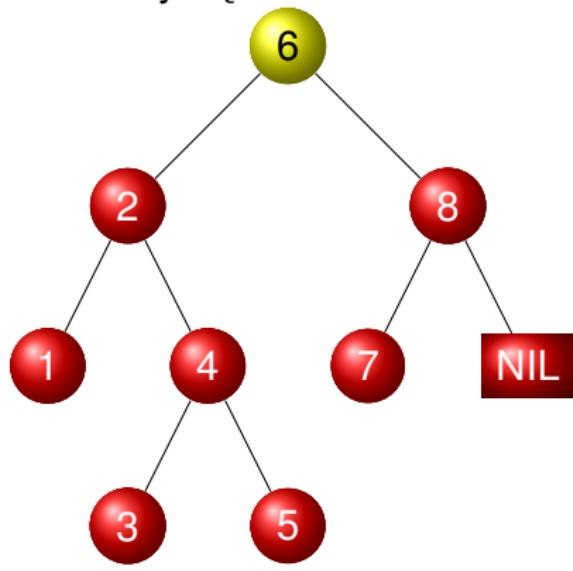
Usuwany węzeł: 6

Usuwany węzeł: 6

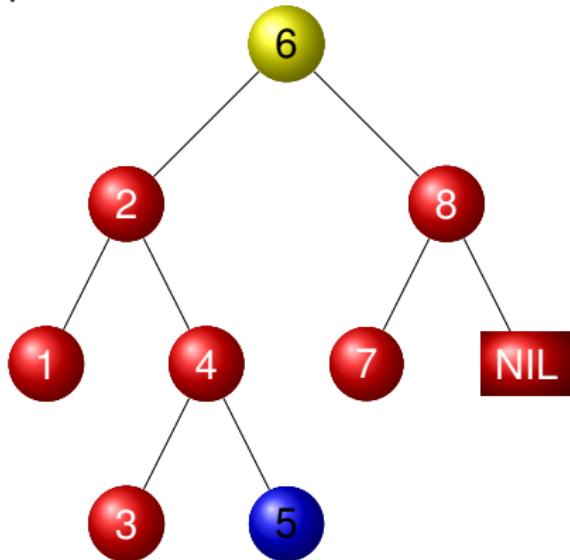


Usuwany węzeł: 6

Usuwany węzeł: 6

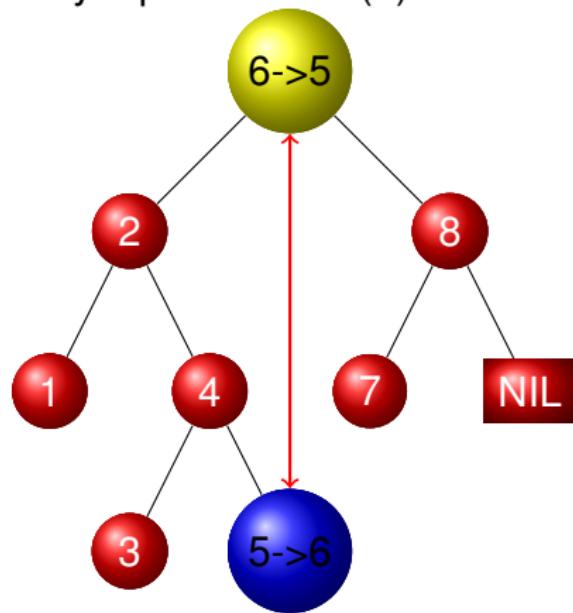


Najpierw szukamy elementu maksymalnego w lewym poddrzewie. Jest nim 5.



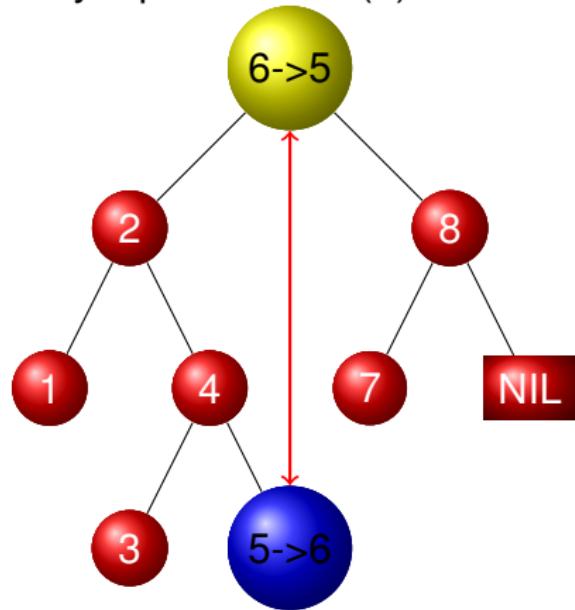
Usuwany węzeł: 6

Następnie zamieniamy usuwany element (6) ze znalezionym elementem maksymalnym w lewym poddrzewie (5).

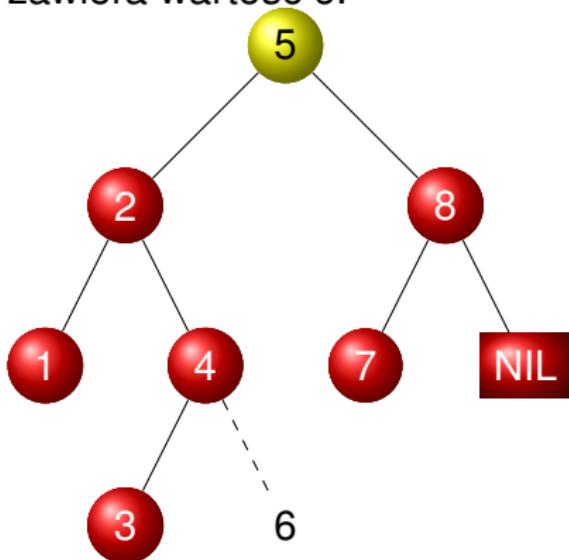


Usuwany węzeł: 6

Następnie zamieniamy usuwany element (6) ze znalezionym elementem maksymalnym w lewym poddrzewie (5).



Usuwamy węzeł, który zawierał element maksymalny (5) w lewym poddrzewie. Teraz węzeł ten zawiera wartość 6.



Algorytmy i Struktury Danych

Kopce

dr hab. Bożena Woźna-Szcześniak
bwozna@gmail.com

Jan Długosz University, Poland

Wykład 13



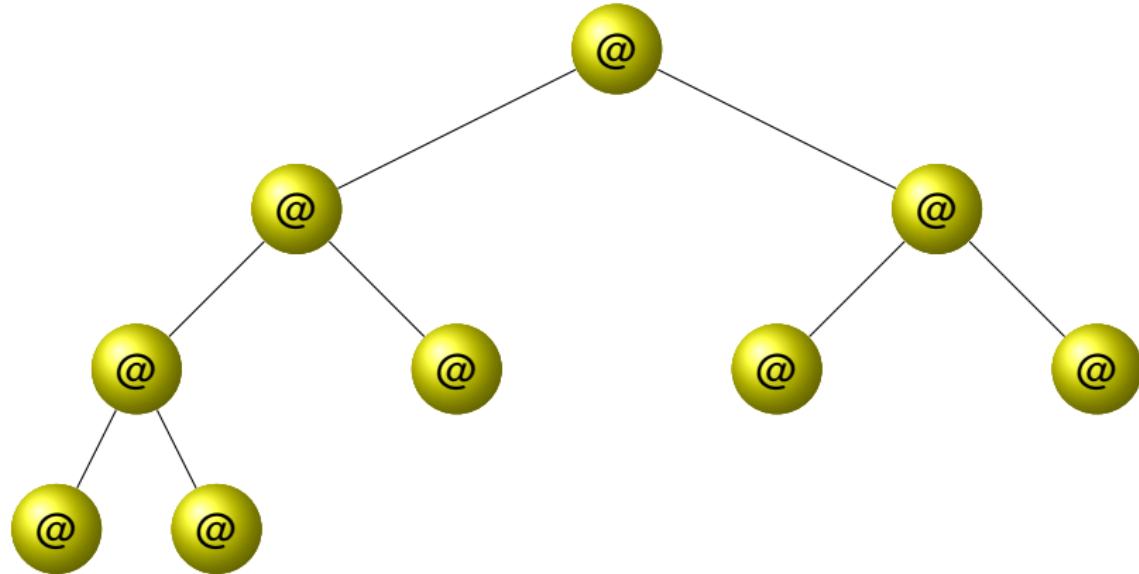
Plan wykładu

- Drzewa zrównoważone
- Kopiec (sterta)
 - Definicja
 - Operacja dodawania i usuwania elementu z kopca
 - Dynamiczne realizacje
- Sortowanie przez kopcowanie
- Implementacja kopca w tablicy
- Kolejki priorytetowe

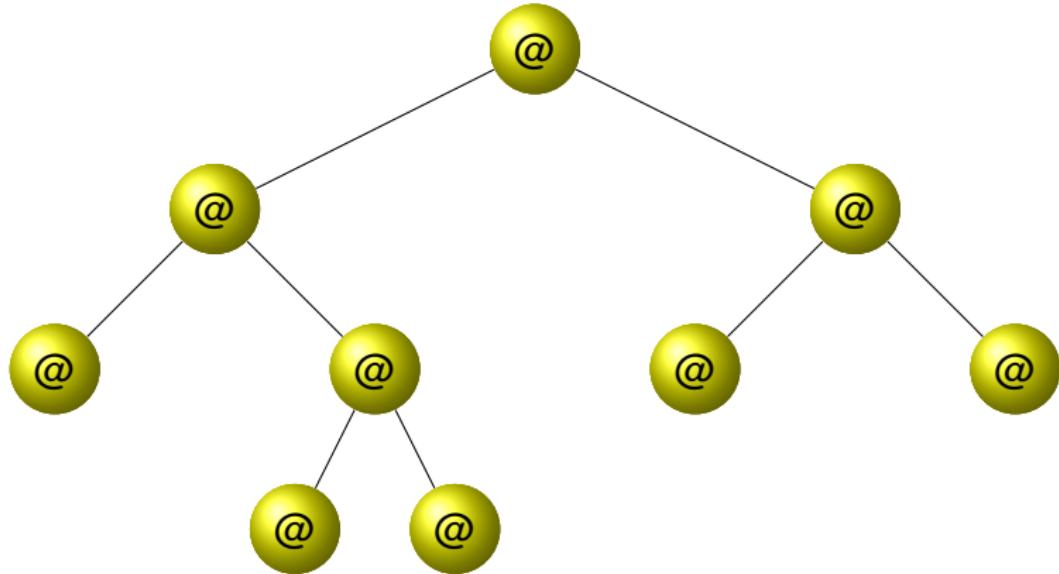
Drzewa zrównoważone

- Drzewo (binarne) jest **zrównoważone**, jeżeli dla każdego węzła wysokości dwóch jego poddrzew (lewego i prawego) różnią się co najwyżej o 1 (własność tzw. drzew AVL)
- Dla drzewa zrównoważonego o liczbie węzłów równej n każda droga od korzenia do jednego z węzłów (w tym liści) nie jest dłuższa niż $\log_2(n)$.

Zrównoważonego drzewa binarne - przykład 1



Zrównoważonego drzewa binarne - przykład 2

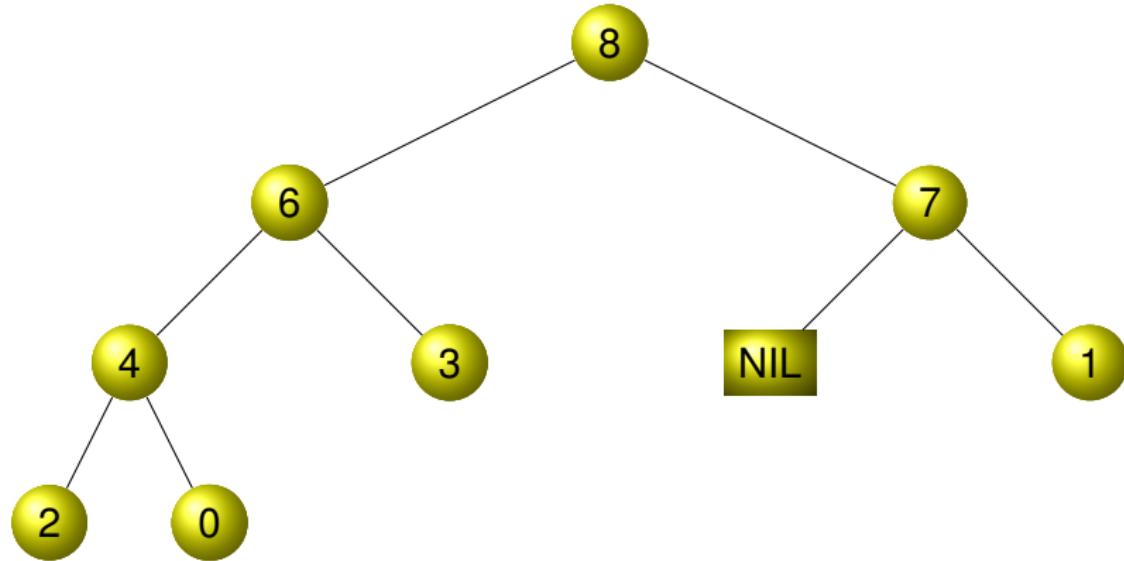


Drzewa częściowo uporządkowane

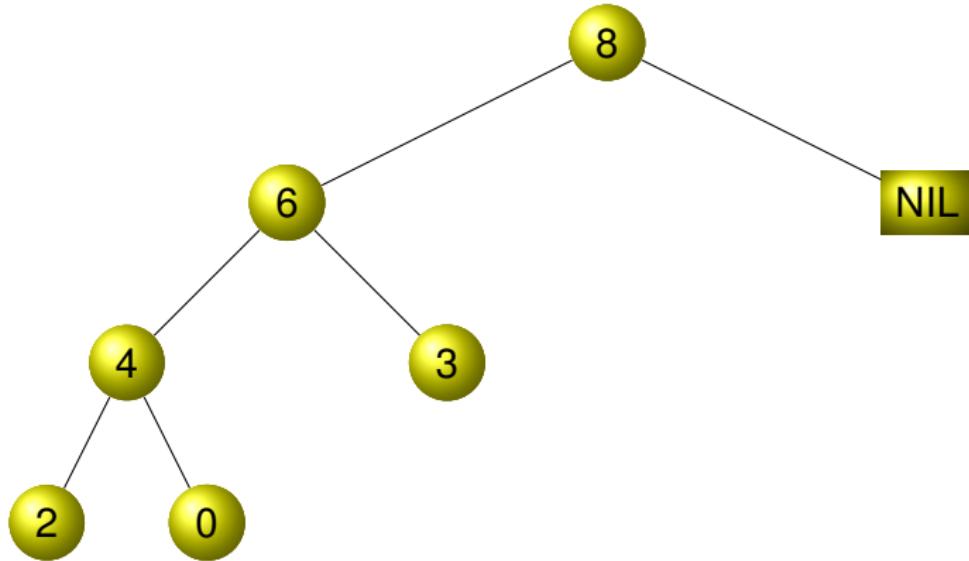
Drzewa częściowo uporządkowane (ang. Partially ordered tree) to drzewa binarne o następującej własności:

- Element przechowywany w węźle musi mieć co najmniej (co najwyżej) tak dużą wartość, jak wartości następników tego węzła.
- Własność ta oznacza, że element w korzeniu dowolnego poddrzewa jest zawsze największym (najmniejszym) elementem tego poddrzewa.

Drzewo częściowo uporządkowane - przykład 1

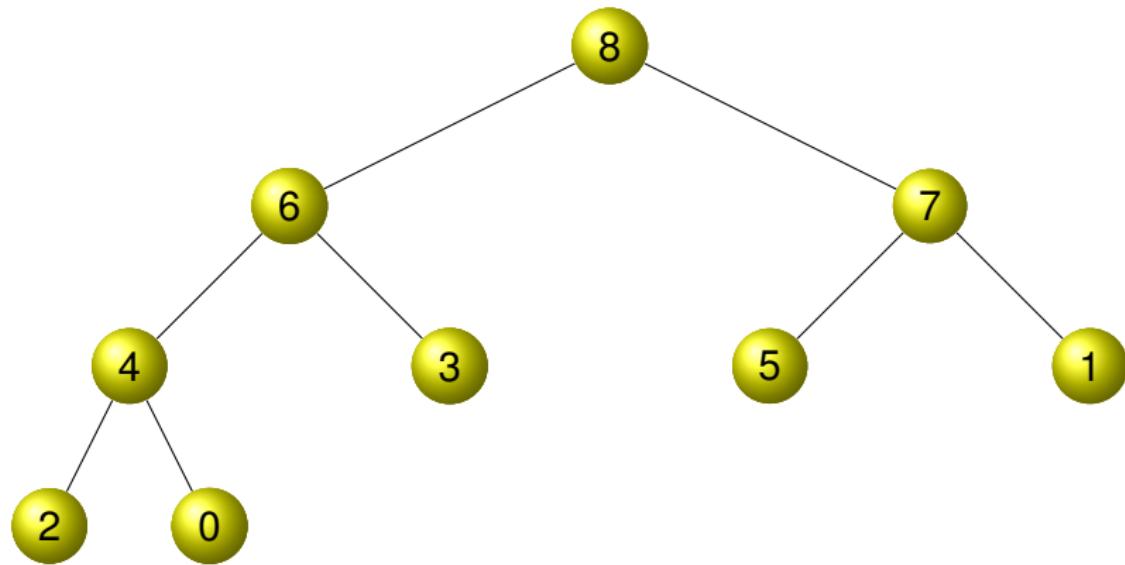


Drzewo częściowo uporządkowane - przykład 2



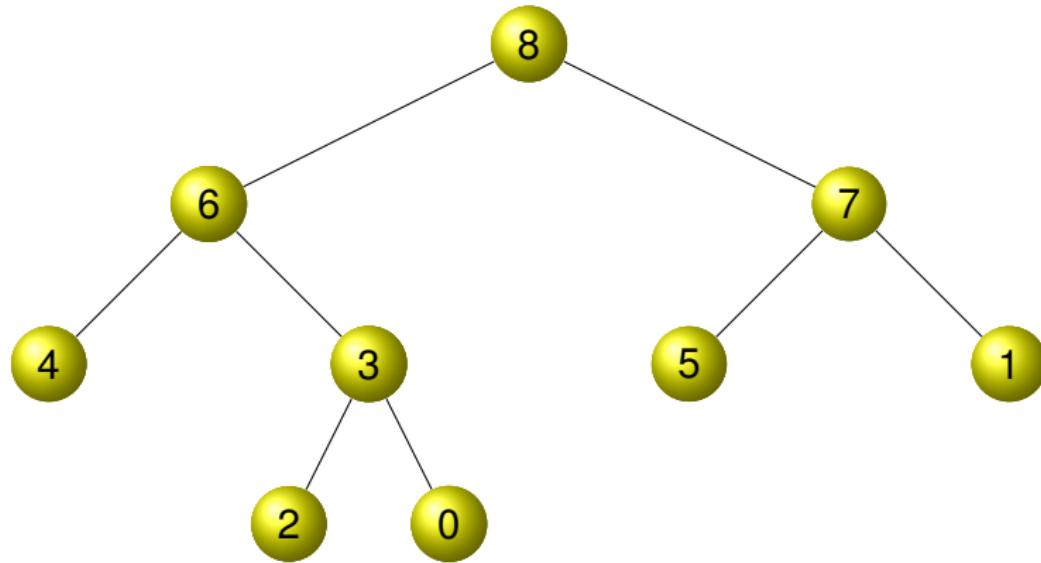
Drzewo częściowo uporządkowane i zrównoważone - przykład 1

Drzewo częściowo uporządkowane jest zrównoważone, jeżeli jest drzewem zrównoważonym.



Drzewo częściowo uporządkowane i zrównoważone - przykład 2

Drzewo częściowo uporządkowane jest zrównoważone, jeżeli jest drzewem zrównoważonym.

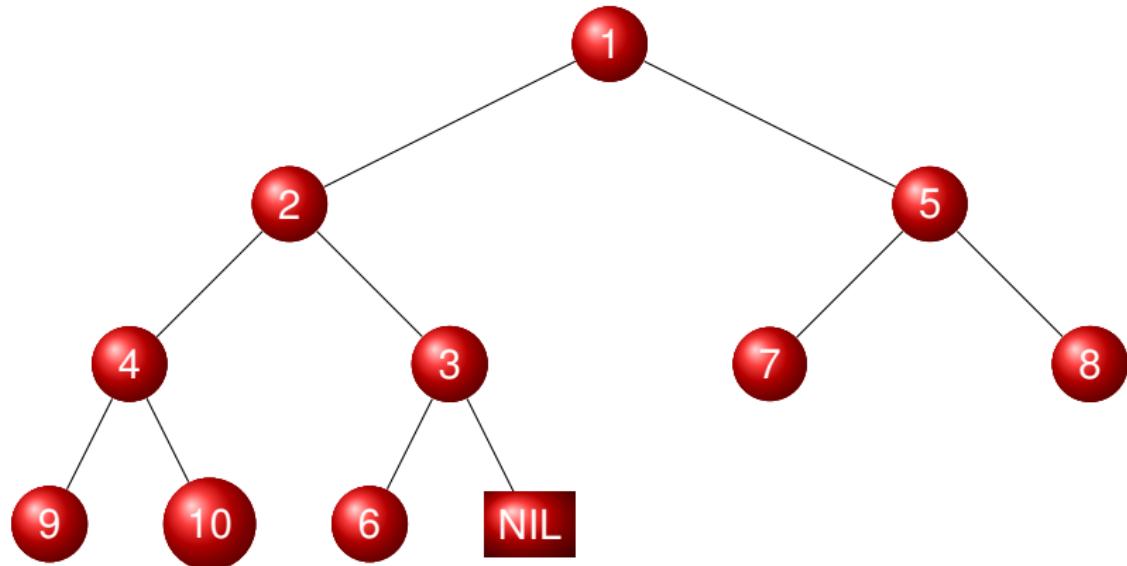


Kopiec (ang. Heap)

Drzewo binarne jest **kopcem (stertą, stogiem)** jeżeli jest:

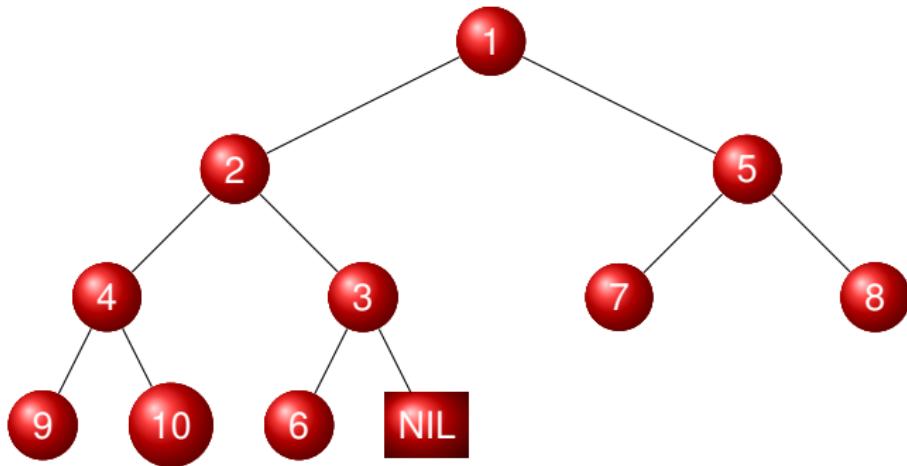
- **częściowo uporządkowane**, czyli wartości przechowywane w następcach każdego węzła są mniejsze od wartości w danym węźle (tzw. **kopiec maksymalny**) lub wartości przechowywane w następcach każdego węzła są większe od wartości w danym węźle (tzw. **kopiec minimalny**).
- **doskonałe**, czyli zrównoważone i wszystkie liście najniższego poziomu znajdują się na jego skrajnych, lewych pozycjach.

Kopiec minimalny - przykład

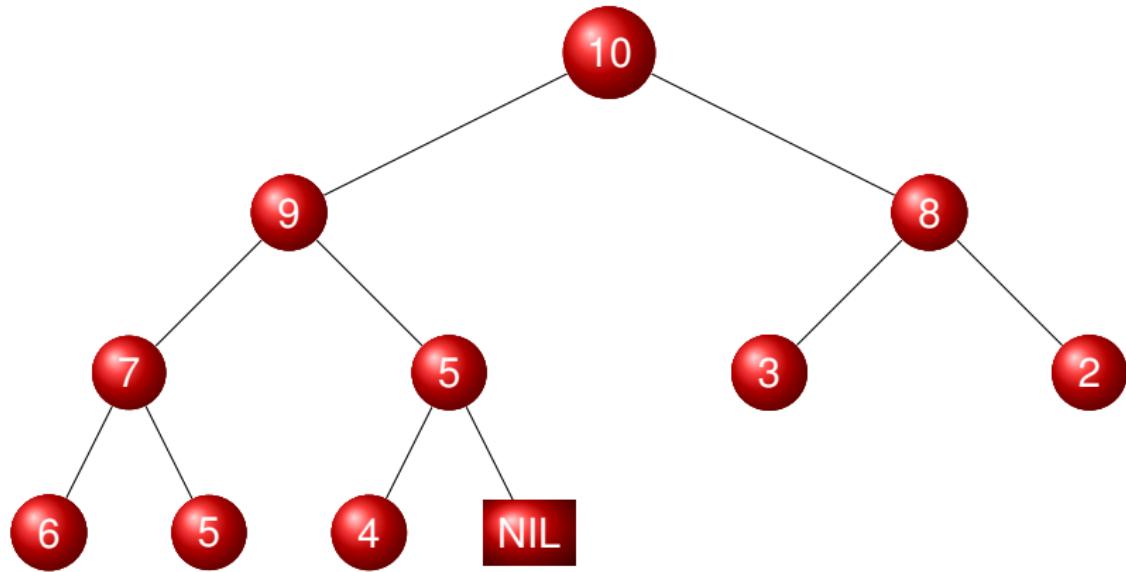


Kopiec minimalny - własności

- Etykiety na dowolnej drodze od korzenia do liścia tworzą ciąg uporządkowany rosnąco.
- Element najmniejszy wśród etykiet wierzchołków znajduje się w korzeniu drzewa.

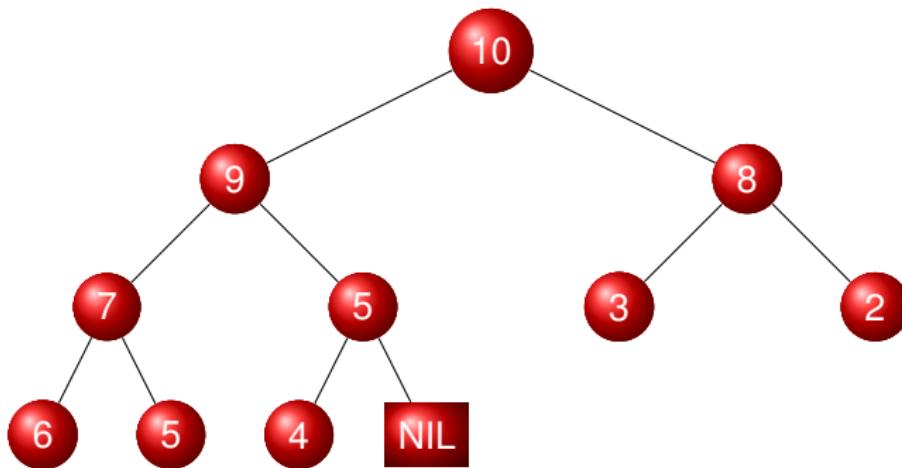


Kopiec maksymalny - przykład



Kopiec maksymalny - własności

- Etykiety na dowolnej drodze od korzenia do liścia tworzą ciąg uporządkowany malejąco.
- Element największy wśród etykiet wierzchołków znajduje się w korzeniu drzewa.



Podstawowe operacje na kopcach

- Dodawanie nowego elementu do kopca
- Usunięcie korzenia z kopca.
- Przechodzenie po kopcu (metoda wszerz (BFS), metoda w głąb (DFS)) - realizacje identyczne do tych na drzewach poszukiwań binarnych.

Dodawanie nowego elementu do kopca

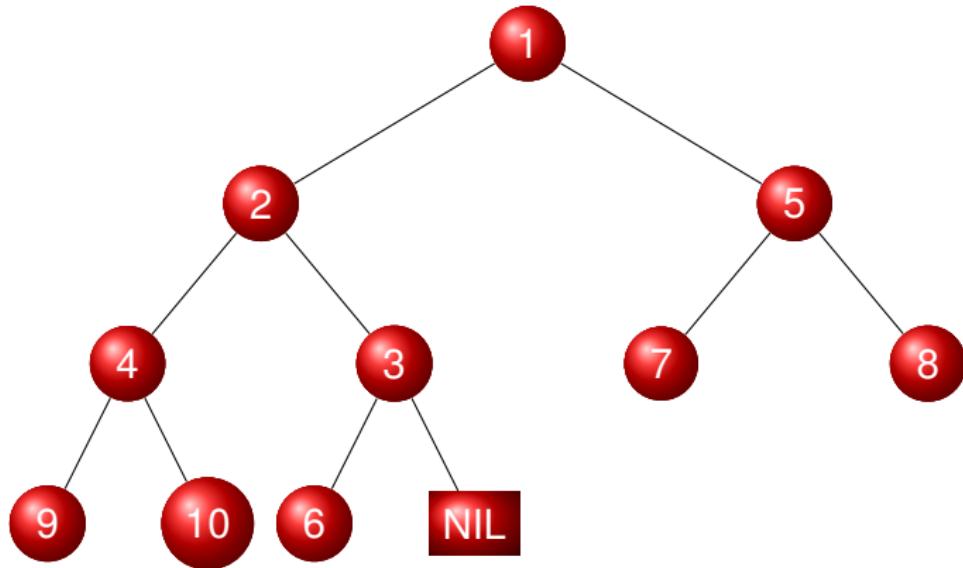
- **Cel: dodanie nowego elementu do kopca.**
- **Dane wejściowe:**
 - Dowiązanie do korzenia drzewa 'Root';
 - Nowe dane elementarne;

Wstawienie nowego elementu do kopca minimalnego - algorytm

- Utwórz nowy element x i ustal dane elementarne;
- Dowiąż nowy wierzchołek x do pierwszego z lewej wierzchołka na przedostatnim poziomie drzewa, którego rząd jest < 2 .
- Jeżeli tak otrzymane drzewo nie jest częściowo uporządkowane, to przechodząc wzduż drogi od liścia x do korzenia, poprawić etykiety zamieniając etykietę ojca z etykietą syna, jeśli etykieta ojca jest **większa** niż etykieta syna.

Wstawienie nowego elementu do kopca minimalnego - idea

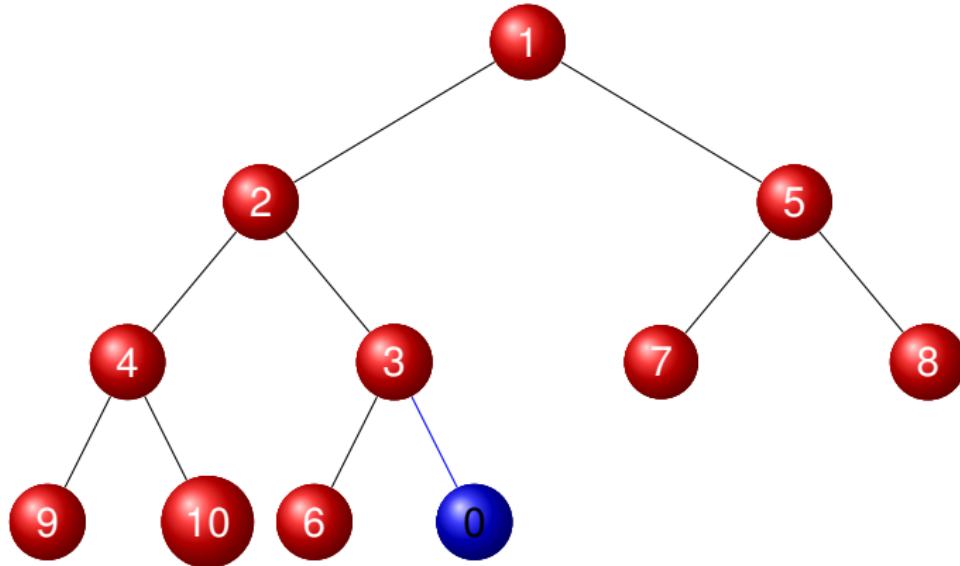
Krok 1: Kopiec początkowy.



Wstawienie nowego elementu do kopca minimalnego - idea

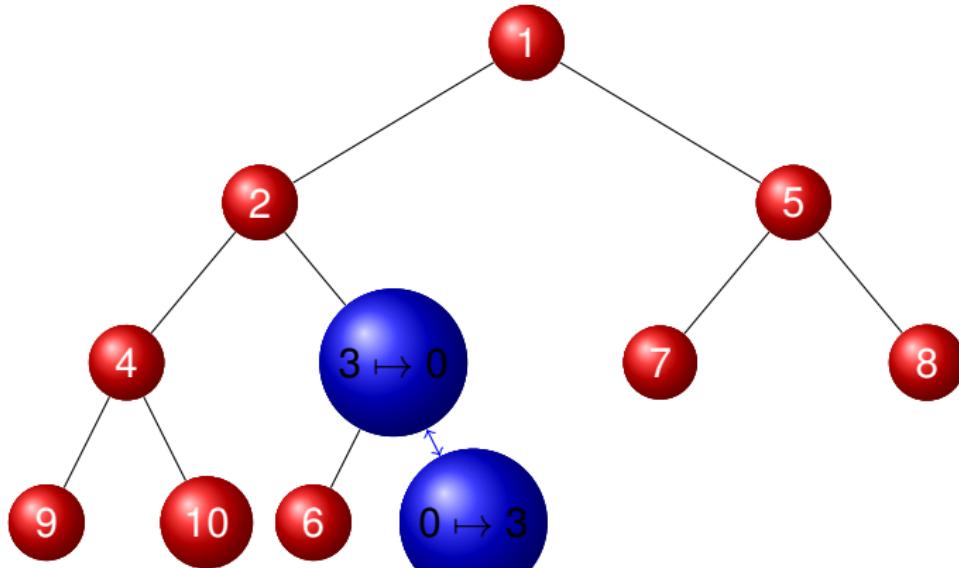
Krok 2:

- Utwórz nowy element x (np. 0) i ustal dane elementarne;
- Dowiąż nowy wierzchołek x do pierwszego z lewej wierzchołka na przedostatnim poziomie drzewa, którego rząd jest < 2 .



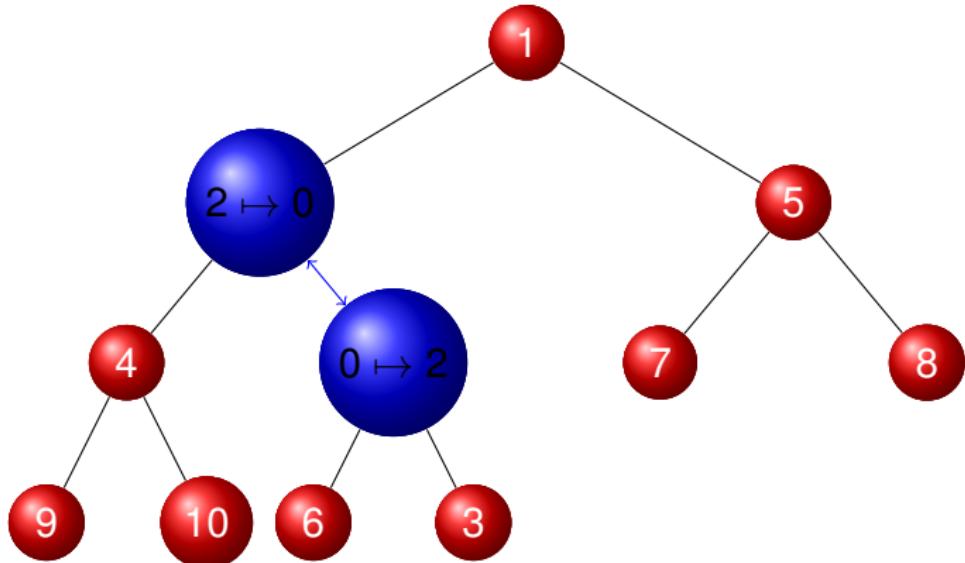
Wstawienie nowego elementu do kopca minimalnego - idea

Krok 3a: Jeżeli tak otrzymane drzewo nie jest częściowo uporządkowane, to przechodząc wzdłuż drogi od liścia x do korzenia, poprawić etykiety zamieniając etykietę ojca z etykietą syna, jeśli etykieta ojca jest **większa** niż etykieta syna.



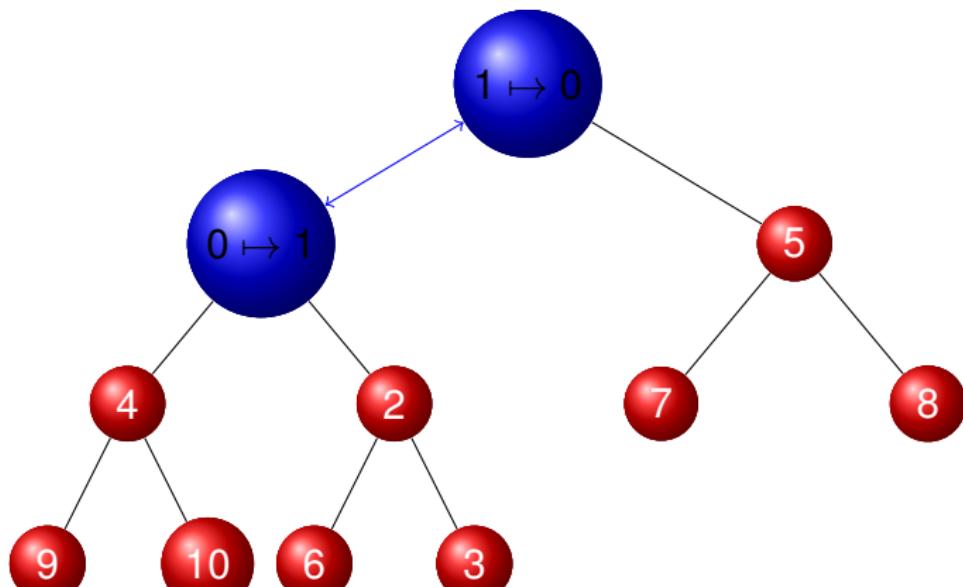
Wstawienie nowego elementu do kopca minimalnego - idea

Krok 3b: Jeżeli tak otrzymane drzewo nie jest częściowo uporządkowane, to przechodząc wzdłuż drogi od liścia x do korzenia, poprawić etykiety zamieniając etykietę ojca z etykietą syna, jeśli etykieta ojca jest **większa** niż etykieta syna.



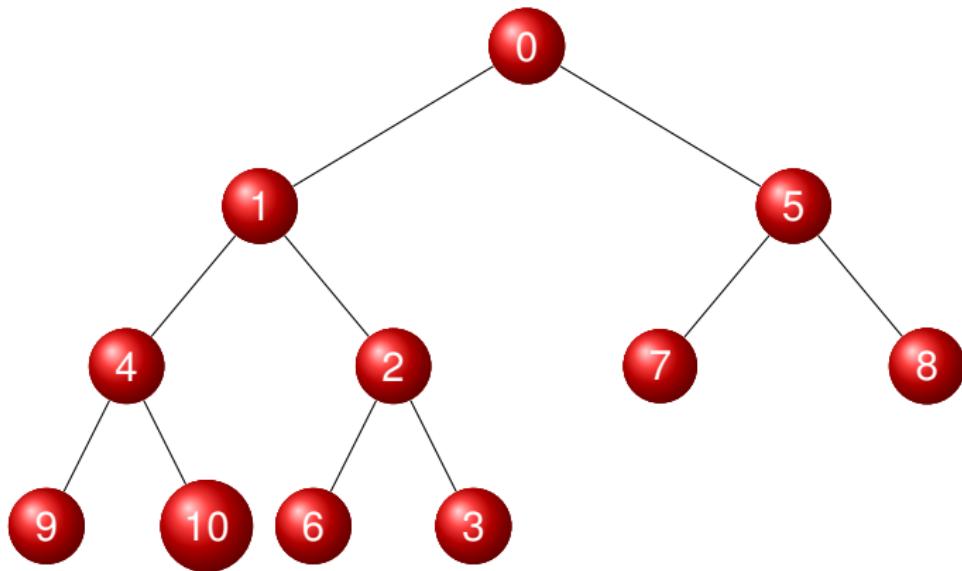
Wstawienie nowego elementu do kopca minimalnego - idea

Krok 3c: Jeżeli tak otrzymane drzewo nie jest częściowo uporządkowane, to przechodząc wzdłuż drogi od liścia x do korzenia, poprawić etykiety zamieniając etykietę ojca z etykietą syna, jeśli etykieta ojca jest **większa** niż etykieta syna.



Wstawienie nowego elementu do kopca minimalnego - idea

Efekt końcowy.

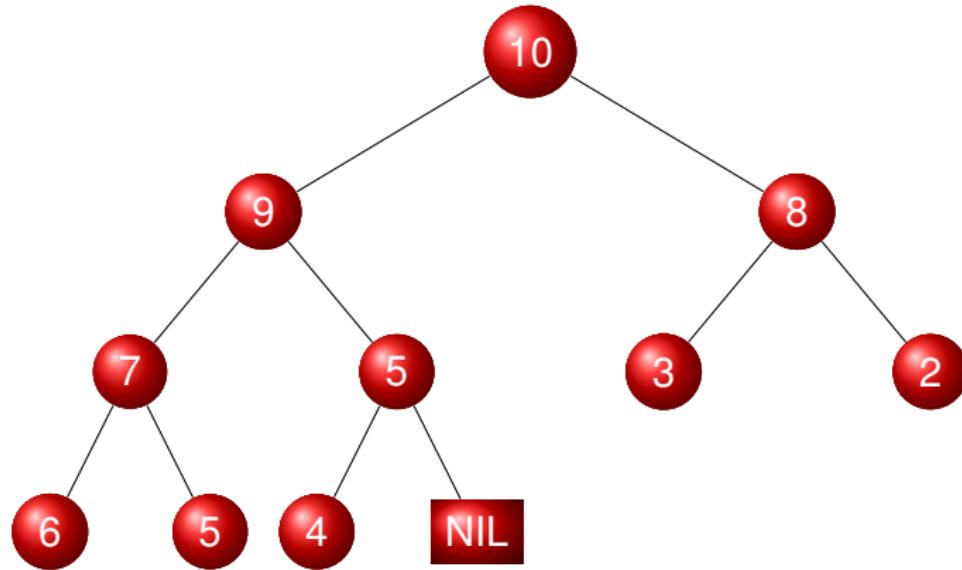


Wstawienie nowego elementu do kopca maksymalnego - algorytm

- Utwórz nowy element x i ustal dane elementarne;
- Dowiąż nowy wierzchołek x do pierwszego z lewej wierzchołka na przedostatnim poziomie drzewa, którego rząd jest < 2 .
- Jeżeli tak otrzymane drzewo nie jest częściowo uporządkowane, to przechodząc wzdłuż drogi od liścia x do korzenia, poprawić etykiety zamieniając etykietę ojca z etykietą syna, jeśli etykieta ojca jest **mniejsza** niż etykieta syna.

Wstawienie nowego elementu do kopca maksymalnego - idea

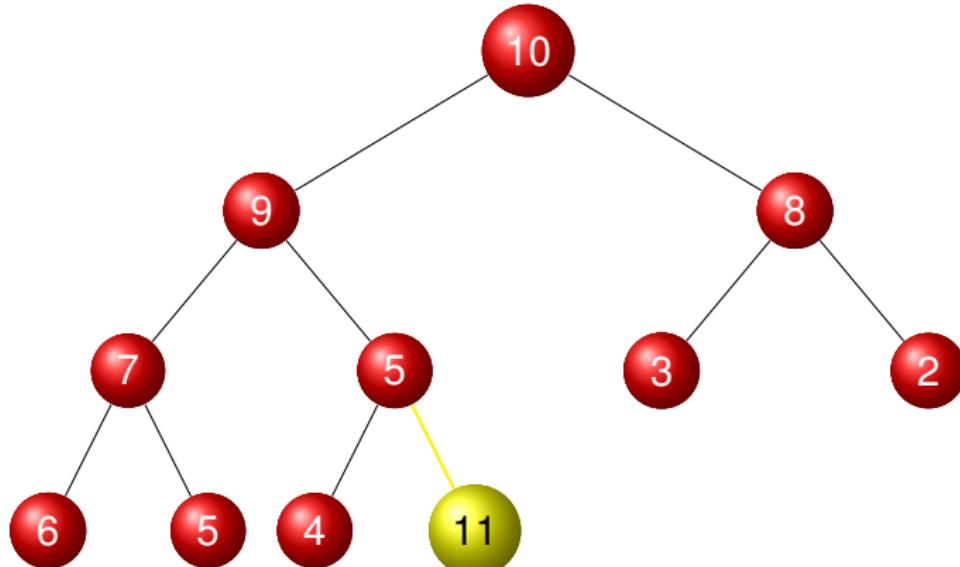
Krok 1: Kopiec początkowy.



Wstawienie nowego elementu do kopca maksymalnego - idea

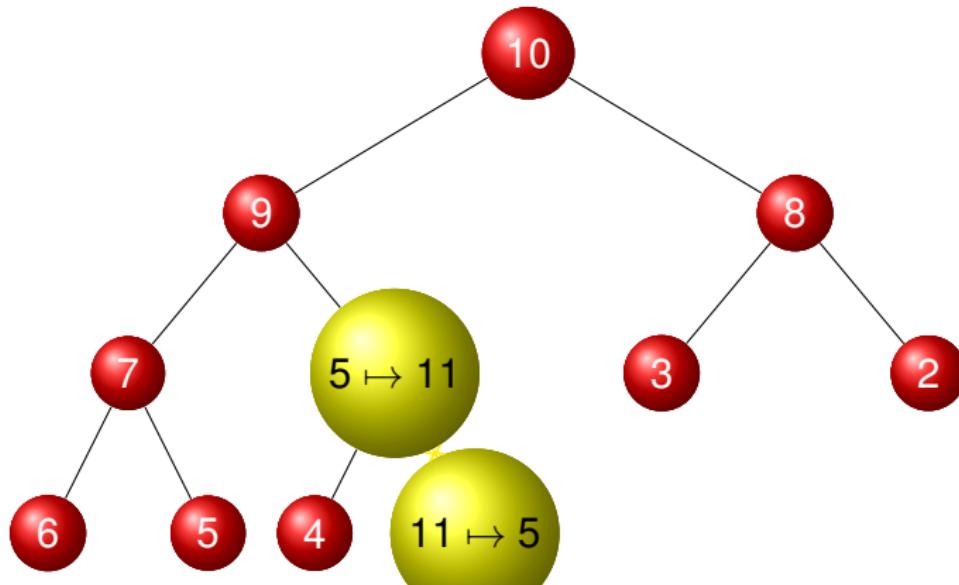
Krok 2:

- Utwórz nowy element x (np. 11) i ustal dane elementarne;
- Dowiąż nowy wierzchołek x do pierwszego z lewej wierzchołka na przedostatnim poziomie drzewa, którego rząd jest < 2 .



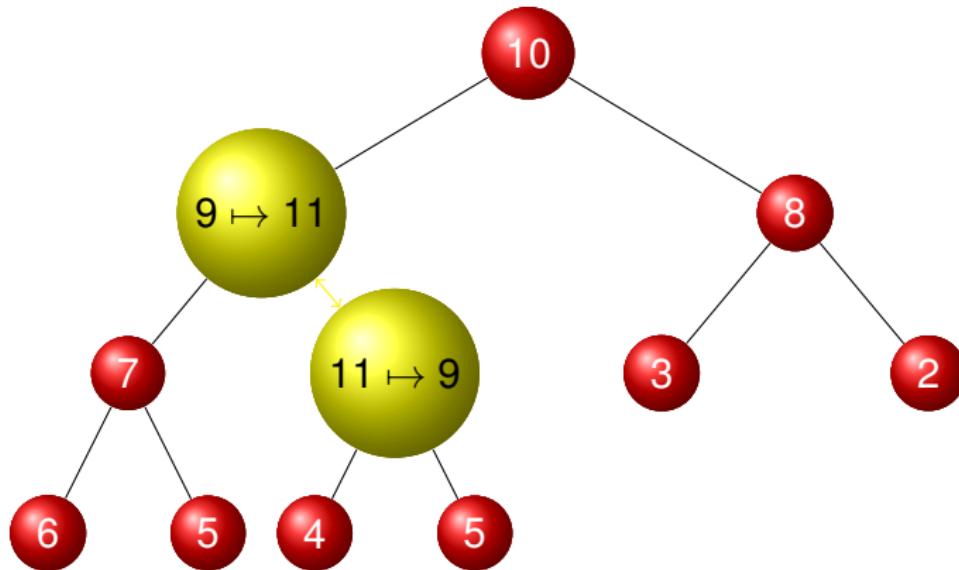
Wstawienie nowego elementu do kopca maksymalnego - idea

Krok 3a: Jeżeli tak otrzymane drzewo nie jest częściowo uporządkowane, to przechodząc wzduż drogi od liścia x do korzenia, poprawić etykiety zamieniając etykietę ojca z etykietą syna, jeśli etykieta ojca jest **większa** niż etykieta syna.



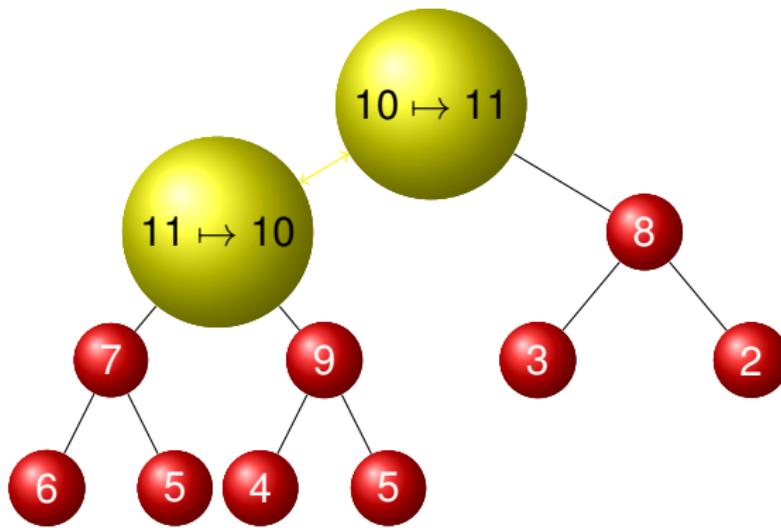
Wstawienie nowego elementu do kopca maksymalnego - idea

Krok 3b: Jeżeli tak otrzymane drzewo nie jest częściowo uporządkowane, to przechodząc wzdłuż drogi od liścia x do korzenia, poprawić etykiety zamieniając etykietę ojca z etykietą syna, jeśli etykieta ojca jest **większa** niż etykieta syna.



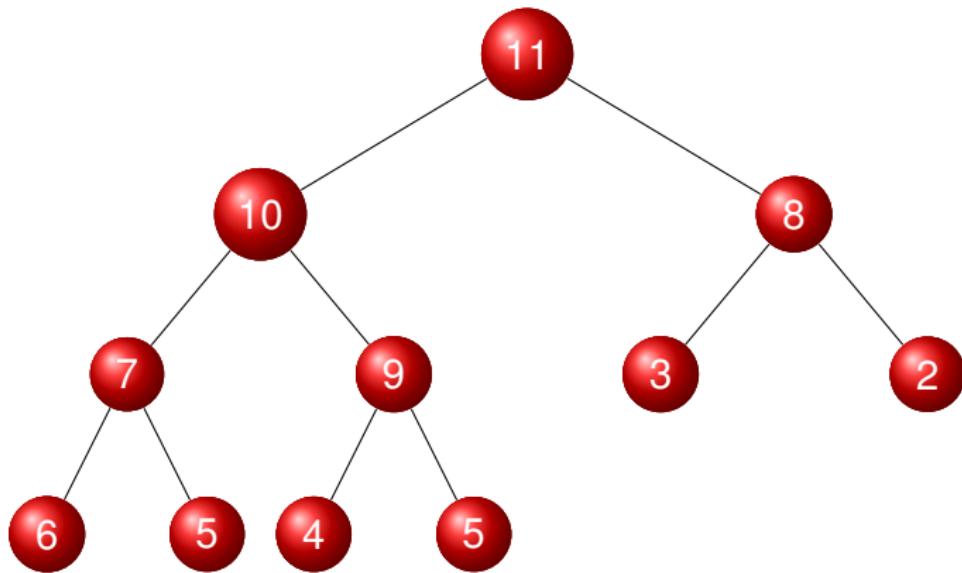
Wstawienie nowego elementu do kopca maksymalnego - idea

Krok 3c: Jeżeli tak otrzymane drzewo nie jest częściowo uporządkowane, to przechodząc wzdłuż drogi od liścia x do korzenia, poprawić etykiety zamieniając etykietę ojca z etykietą syna, jeśli etykieta ojca jest **większa** niż etykieta syna.



Wstawienie nowego elementu do kopca maksymalnego - idea

Efekt końcowy.



Usuwanie korzenia z kopca

- **Cel:** usunięcie elementu minimalnego (maksymalnego) z kopca;
- **Dane wejściowe:** Dowiązanie do korzenia drzewa “Root”;

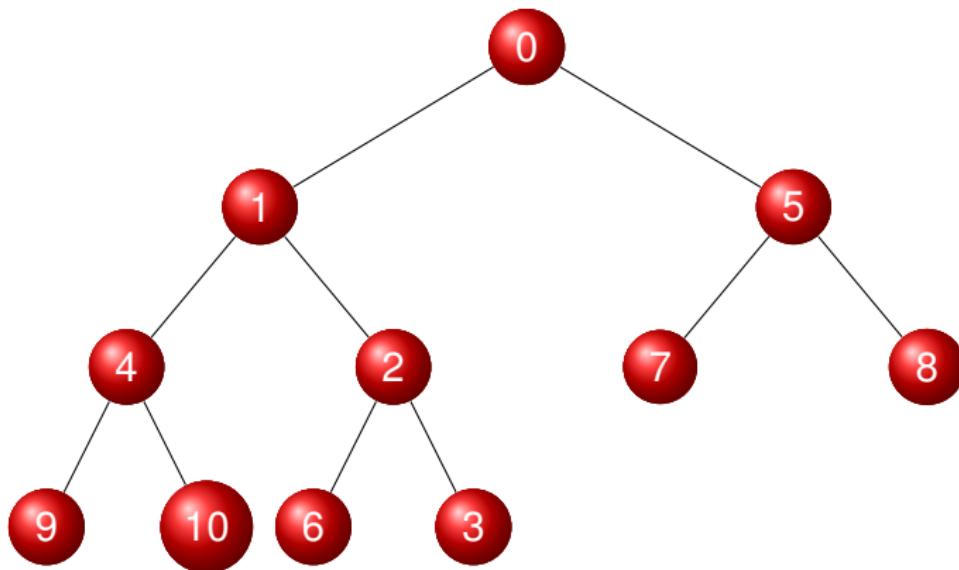
Usuwanie korzenia z kopca minimalnego - algorytm

Niech e będzie etykietą liścia x znajdującego się najbardziej na prawo na ostatnim poziomie kopca.

- Zastąpić etykietę w korzeniu drzewa przez e .
- Usunąć wierzchołek x z drzewa.
- Jeśli tak otrzymane drzewo nie jest kopcem, to zaczynając od korzenia i idąc w kierunku liścia, zamieniać etykietę ojca z etykietą tego z jego synów, którego etykieta ma mniejszą wartość tak długo, aż zostanie otrzymane drzewo częściowo uporządkowane.

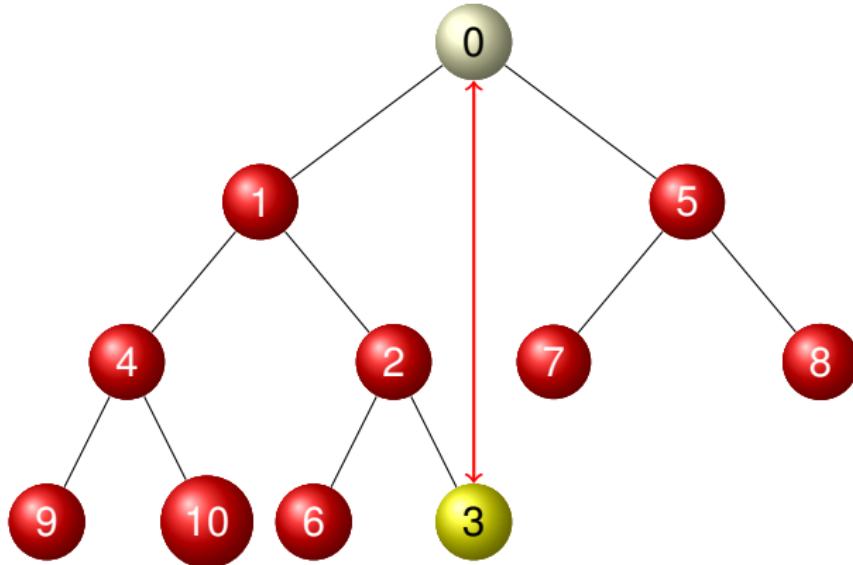
Usuwanie korzenia z kopca minimalnego - idea

Kopiec początkowy.



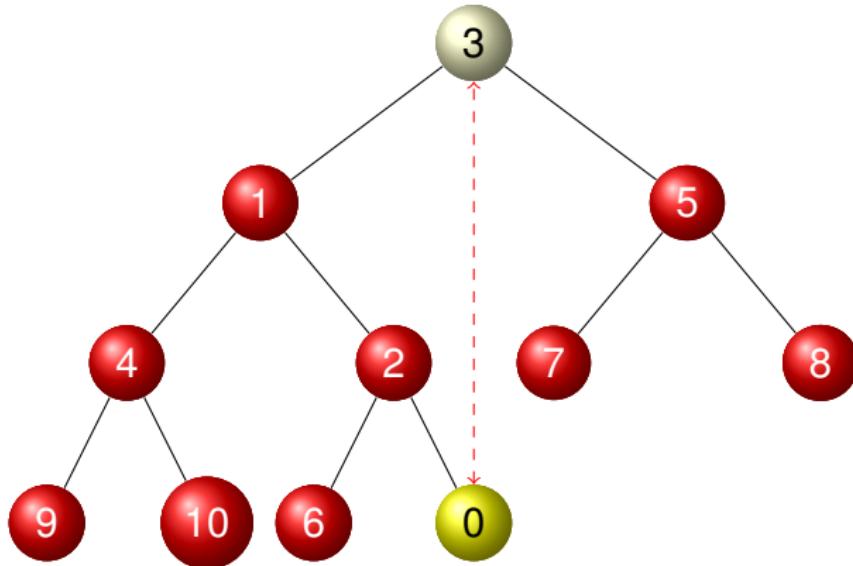
Usuwanie korzenia z kopca minimalnego - idea

Krok 1a: Zastąpić etykietę w korzeniu drzewa przez $e = 3$.



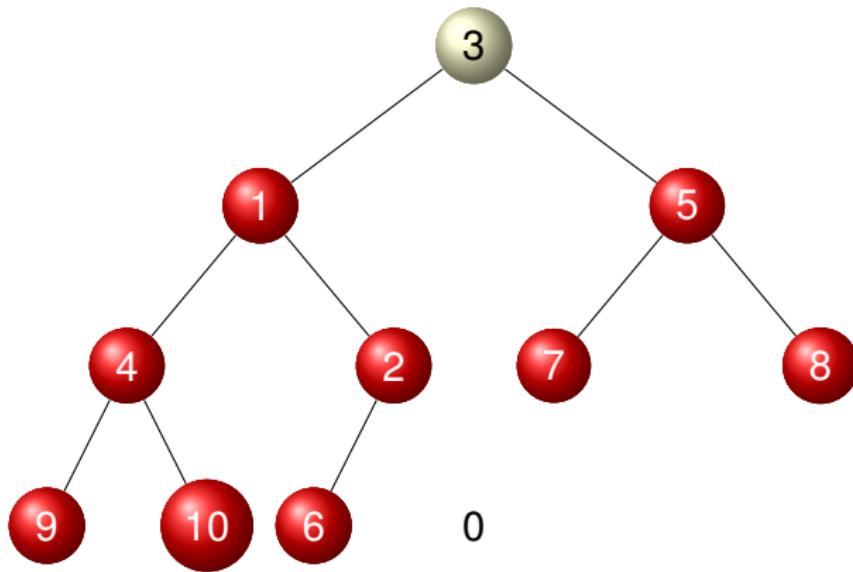
Usuwanie korzenia z kopca minimalnego - idea

Krok 1b: Zastąpić etykietę w korzeniu drzewa przez $e = 3$.



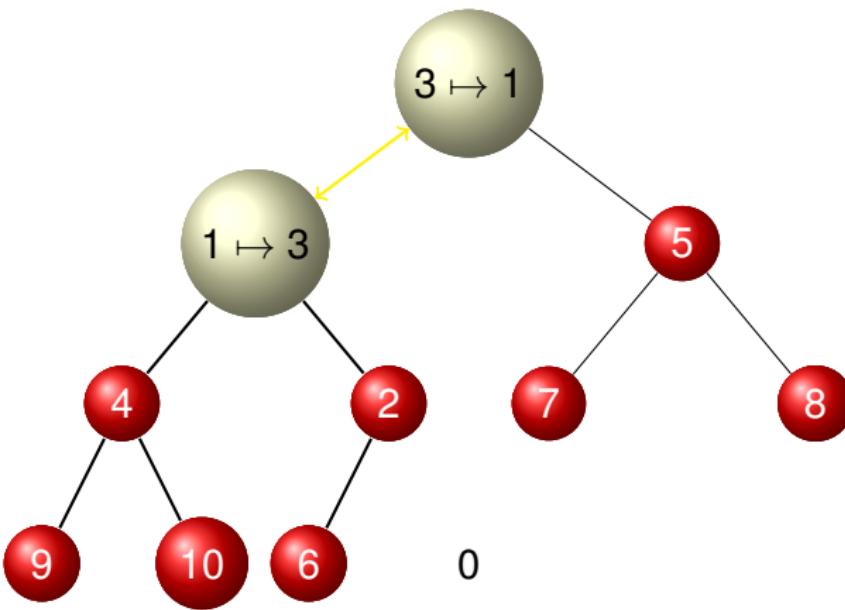
Usuwanie korzenia z kopca minimalnego - idea

Krok 2: Usunąć wierzchołek x (u nas zawiera on stary korzeń-0) z drzewa.



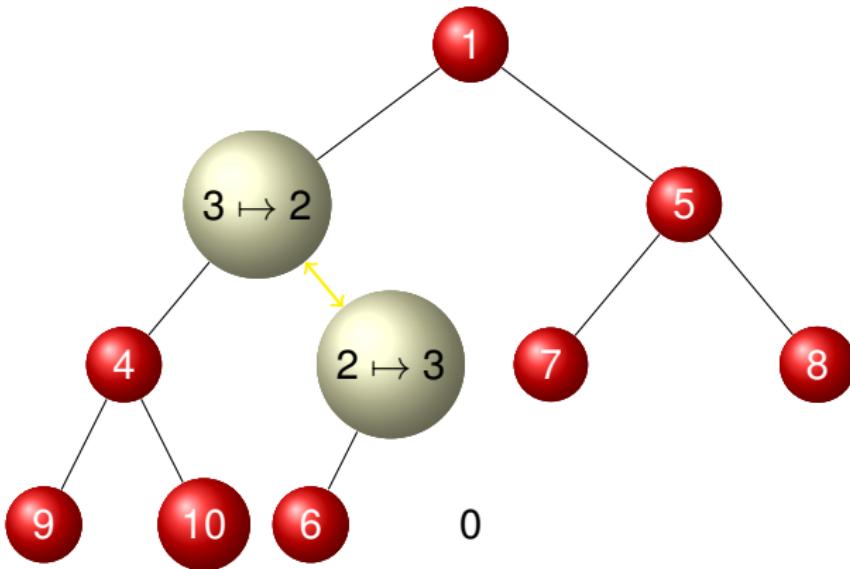
Usuwanie korzenia z kopca minimalnego - idea

Krok 3a: Jeśli tak otrzymane drzewo nie jest kopcem, to zaczynając od korzenia i idąc w kierunku liścia, zamieniać etykietę ojca z etykietą tego z jego synów, którego etykieta ma mniejszą wartość, tak długo aż zostanie otrzymane drzewo częściowo uporządkowane.



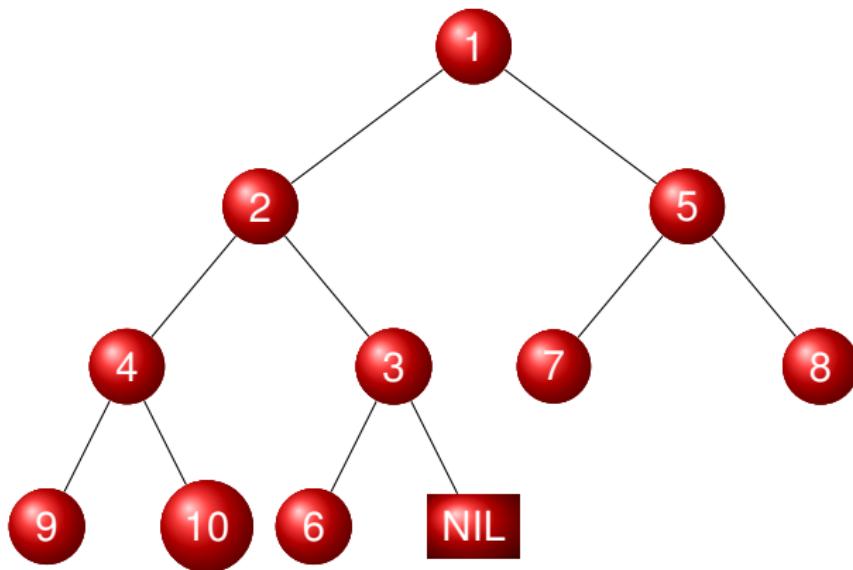
Usuwanie korzenia z kopca minimalnego - idea

Krok 3b: Jeśli tak otrzymane drzewo nie jest kopcem, to zaczynając od korzenia i idąc w kierunku liścia, zamieniać etykietę ojca z etykietą tego z jego synów, którego etykieta ma mniejszą wartość, tak długo aż zostanie otrzymane drzewo częściowo uporządkowane.



Usuwanie korzenia z kopca minimalnego - idea

Efekt końcowy.



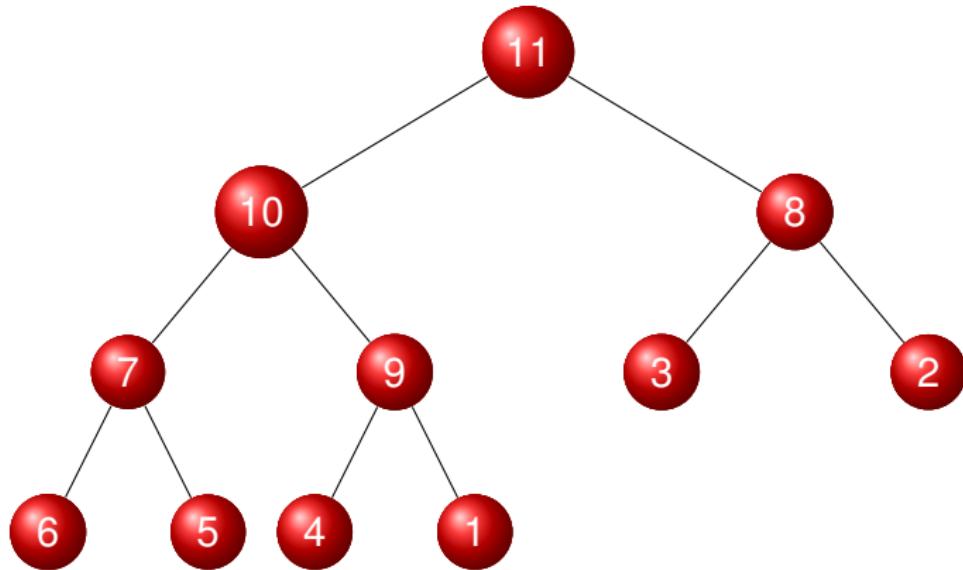
Usuwanie korzenia z kopca maksymalnego - algorytm

Niech e będzie etykietą liścia x znajdującego się najbardziej na prawo na ostatnim poziomie kopca.

- Zastąpić etykietę w korzeniu drzewa przez e .
- Usunąć wierzchołek x z drzewa.
- Jeśli tak otrzymane drzewo nie jest kopcem, to zaczynając od korzenia i idąc w kierunku liścia, zamieniać etykietę ojca z etykietą tego z jego synów, którego etykieta ma większą wartość tak dugo, aż zostanie otrzymane drzewo częściowo uporządkowane.

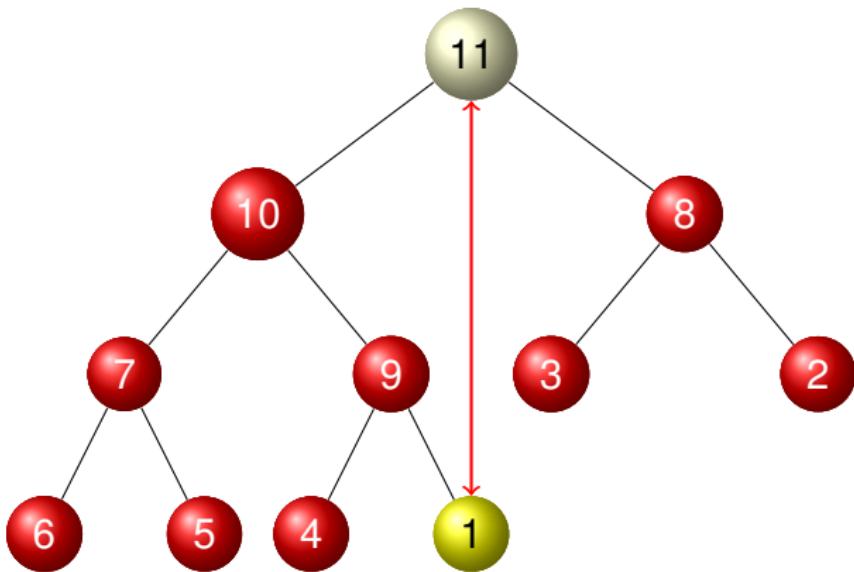
Usuwanie korzenia z kopca maksymalnego - idea

Kopiec początkowy.



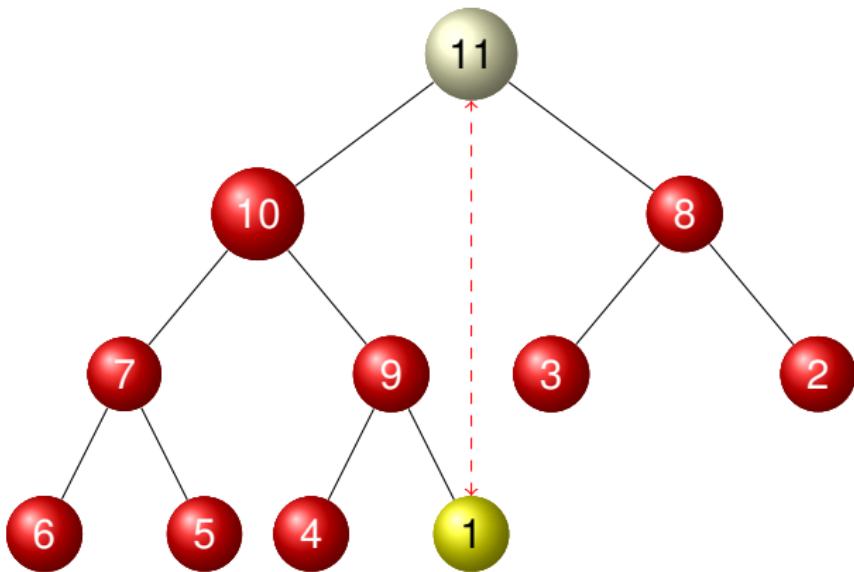
Usuwanie korzenia z kopca maksymalnego - idea

Krok 1a: Zastąpić etykietę w korzeniu drzewa przez $e = 1$.



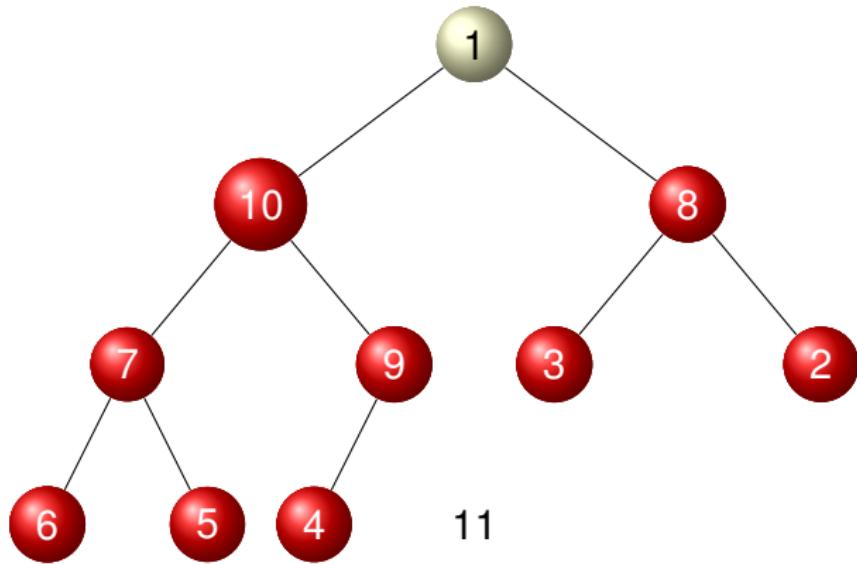
Usuwanie korzenia z kopca maksymalnego - idea

Krok 1b: Zastąpić etykietę w korzeniu drzewa przez $e = 1$.



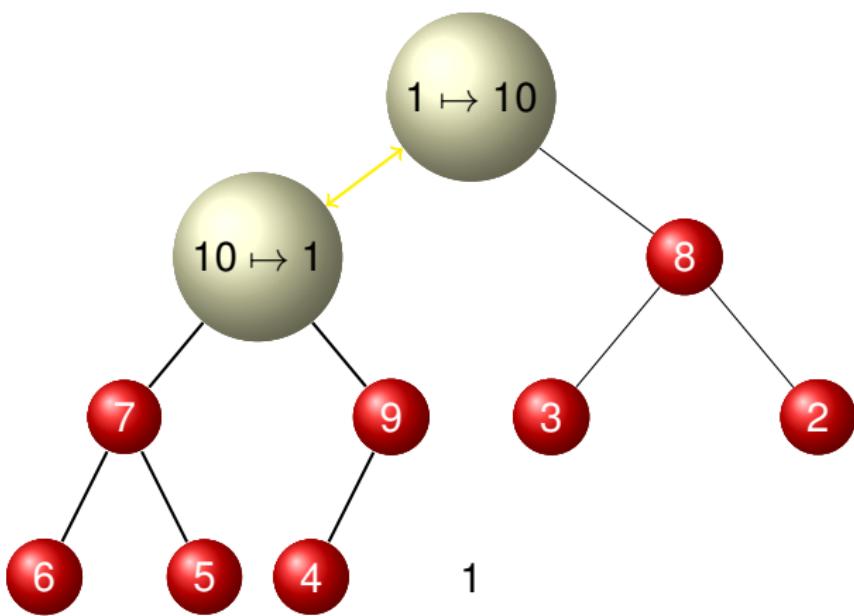
Usuwanie korzenia z kopca maksymalnego - idea

Krok 2: Usunąć wierzchołek x (u nas zawiera on stary korzeń-11) z drzewa.



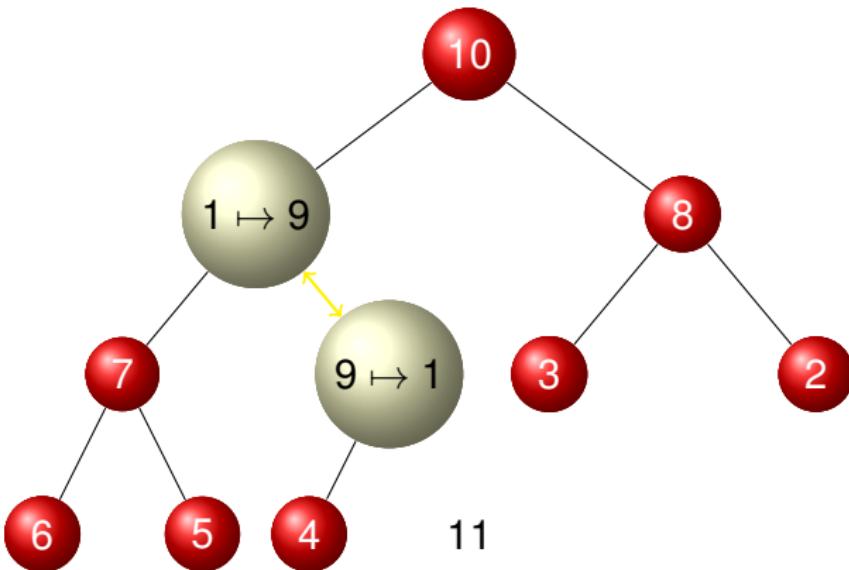
Usuwanie korzenia z kopca maksymalnego - idea

Krok 3a: Jeśli tak otrzymane drzewo nie jest kopcem, to zaczynając od korzenia i idąc w kierunku liścia, zamieniać etykietę ojca z etykietą tego z jego synów, którego etykieta ma większą wartość, tak długo aż zostanie otrzymane drzewo częściowo uporządkowane.



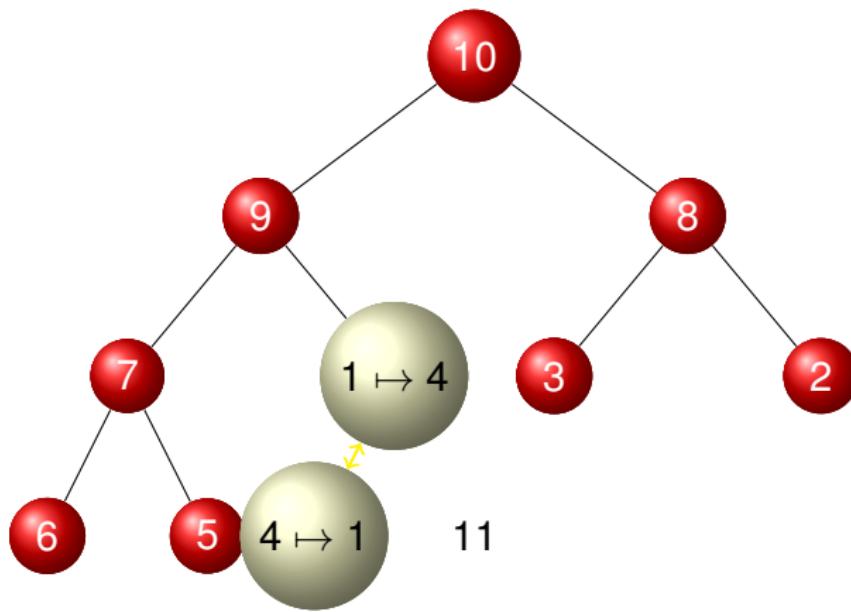
Usuwanie korzenia z kopca maksymalnego - idea

Krok 3b: Jeśli tak otrzymane drzewo nie jest kopcem, to zaczynając od korzenia i idąc w kierunku liścia, zamieniać etykietę ojca z etykietą tego z jego synów, którego etykieta ma większą wartość, tak długo aż zostanie otrzymane drzewo częściowo uporządkowane.



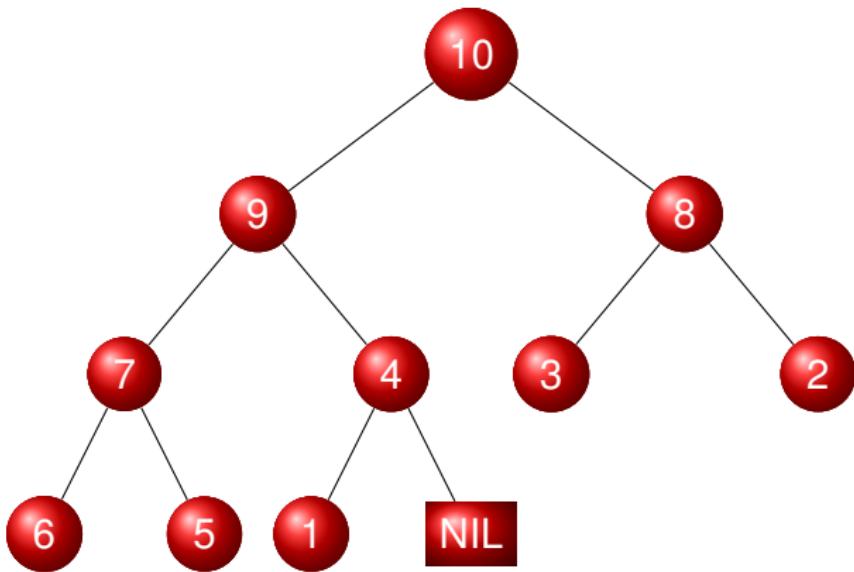
Usuwanie korzenia z kopca maksymalnego - idea

Krok 3c: Jeśli tak otrzymane drzewo nie jest kopcem, to zaczynając od korzenia i idąc w kierunku liścia, zamieniać etykietę ojca z etykietą tego z jego synów, którego etykieta ma większą wartość, tak długo aż zostanie otrzymane drzewo częściowo uporządkowane.



Usuwanie korzenia z kopca maksymalnego - idea

Efekt końcowy.



Koszty operacji

- Niech n będzie liczbą węzłów (wierzchołków) w kopcu, a h wysokością kopca.
- Z definicji kopca mamy:

$$\sum_{i=0}^{h-1} 2^i < n \leq \sum_{i=0}^h 2^i$$

- Dalej: $2^{h-1} - 1 < n \leq 2^{h+1} - 1$
- Dalej: $2^{h-1} < n + 1 \leq 2^{h+1}$
- Dalej: $\log_2(n + 1) - 1 < h \leq \log_2(n + 1)$
- Zatem: $h = \lfloor \log_2(n + 1) \rfloor$.

Koszt operacji wstawiania i usuwania wynosi:

$$O(\log_2(n))$$

Algorytm

- Z danego n elementowego zbioru utworzyć kopiec.
- Dopóki kopiec nie jest pusty, wypisywać i usuwać element minimalny (maksymalny) kopca.

Sortowanie przez kopcowanie

Algorytm

- Z danego n elementowego zbioru utworzyć kopiec.
- Dopóki kopiec nie jest pusty, wypisywać i usuwać element minimalny (maksymalny) kopca.

Koszt sortowania

Koszt sortowania = koszt utworzenia kopca + n^* (koszt usuwania korzenia) = $O(n * \log_2(n)) + n * O(\log_2(n)) = O(n * \log_2(n))$

Implementacja kopków

- Kopiec można zaimplementować bazując na tablicy jednowymiarowej (wektorze) o długości n .
- **Uwaga:** każdy kopiec jest tablicą, ale nie każda tablica jest kopcem.

Implementacja kopków

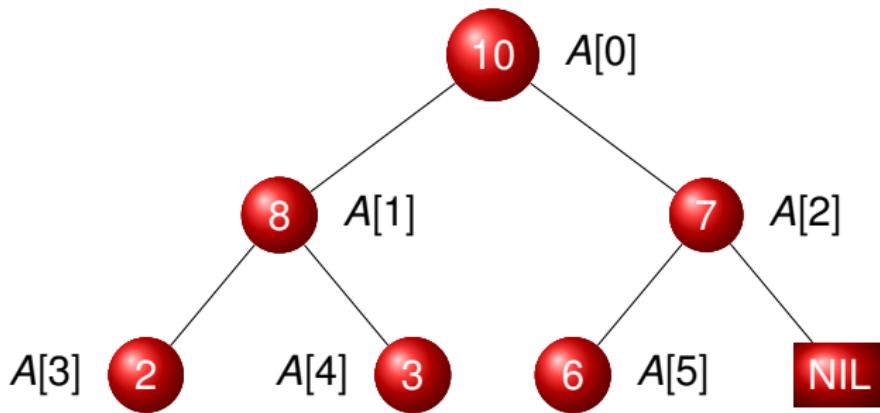
- Kopiec można zaimplementować bazując na tablicy jednowymiarowej (wektorze) o długości n .
- **Uwaga:** każdy kopiec jest tablicą, ale nie każda tablica jest kopcem.
- Kopiec można zaimplementować również w sposób dynamiczny w oparciu o strukturę węzła (dana elementarna, adres lewego poddrzewa, adres prawego poddrzewa).

Tablicowa implementacja kopków

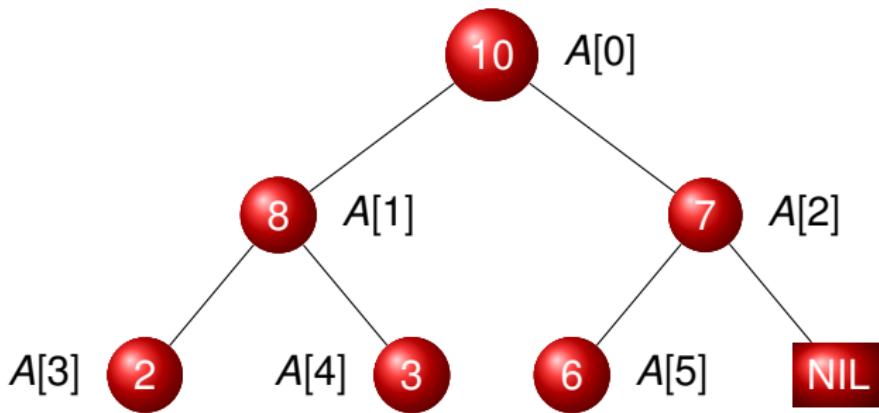
Cechy charakterystyczne tablicy A implementującej kopiec:

- Korzeń znajduje się w $A[0]$.
- Lewy następnik korzenia znajduje się w $A[1]$, prawy następnik korzenia w $A[2]$.
- Ogólnie: lewy następnik węzła zapisanego w $A[i]$ znajduje się w $A[2i + 1]$, prawy następnik - w $A[2i + 2]$ (jeżeli następcy istnieją);

Tablicowa implementacja kopców - przykład



Tablicowa implementacja kopków - przykład



Tablica A:

0	1	2	3	4	5
10	8	7	2	3	6

Zachodzi: $A[i] \geq A[2i + 1]$ oraz $A[i] \geq A[2i + 2]$

Wstawianie do kopca w implementacji tablicowej - przykład

Dany jest kopiec:

5	7	12	10	8	14	19	16	29	11	20
---	---	----	----	---	----	----	----	----	----	----

Wstawianie do kopca w implementacji tablicowej - przykład

Dany jest kopiec:

5	7	12	10	8	14	19	16	29	11	20
---	---	----	----	---	----	----	----	----	----	----

Dodajemy wartość: 3

5	7	12	10	8	14	19	16	29	11	20	3
---	---	----	----	---	----	----	----	----	----	----	---

Wstawianie do kopca w implementacji tablicowej - przykład

Dany jest kopiec:

5	7	12	10	8	14	19	16	29	11	20
---	---	----	----	---	----	----	----	----	----	----

Dodajemy wartość: 3

5	7	12	10	8	14	19	16	29	11	20	3
---	---	----	----	---	----	----	----	----	----	----	---

Naprawiamy kopiec. Krok 1: $3 < - > 14$:

5	7	12	10	8	3	19	16	29	11	20	14
---	---	----	----	---	---	----	----	----	----	----	----

Wstawianie do kopca w implementacji tablicowej - przykład

Dany jest kopiec:

5	7	12	10	8	14	19	16	29	11	20
---	---	----	----	---	----	----	----	----	----	----

Dodajemy wartość: 3

5	7	12	10	8	14	19	16	29	11	20	3
---	---	----	----	---	----	----	----	----	----	----	---

Naprawiamy kopiec. Krok 1: 3 < – > 14:

5	7	12	10	8	3	19	16	29	11	20	14
---	---	----	----	---	---	----	----	----	----	----	----

Naprawiamy kopiec. Krok 2: 3 < – > 12:

5	7	3	10	8	12	19	16	29	11	20	14
---	---	---	----	---	----	----	----	----	----	----	----

Wstawianie do kopca w implementacji tablicowej - przykład

Dany jest kopiec:

5	7	12	10	8	14	19	16	29	11	20
---	---	----	----	---	----	----	----	----	----	----

Dodajemy wartość: 3

5	7	12	10	8	14	19	16	29	11	20	3
---	---	----	----	---	----	----	----	----	----	----	---

Naprawiamy kopiec. Krok 1: $3 < - > 14$:

5	7	12	10	8	3	19	16	29	11	20	14
---	---	----	----	---	---	----	----	----	----	----	----

Naprawiamy kopiec. Krok 2: $3 < - > 12$:

5	7	3	10	8	12	19	16	29	11	20	14
---	---	---	----	---	----	----	----	----	----	----	----

Naprawiamy kopiec. Krok 3: $3 < - > 5$:

3	7	5	10	8	12	19	16	29	11	20	14
---	---	---	----	---	----	----	----	----	----	----	----

Wstawianie do kopca w implementacji tablicowej - przykład

Dany jest kopiec:

5	7	12	10	8	14	19	16	29	11	20
---	---	----	----	---	----	----	----	----	----	----

Dodajemy wartość: 3

5	7	12	10	8	14	19	16	29	11	20	3
---	---	----	----	---	----	----	----	----	----	----	---

Naprawiamy kopiec. Krok 1: $3 < - > 14$:

5	7	12	10	8	3	19	16	29	11	20	14
---	---	----	----	---	---	----	----	----	----	----	----

Naprawiamy kopiec. Krok 2: $3 < - > 12$:

5	7	3	10	8	12	19	16	29	11	20	14
---	---	---	----	---	----	----	----	----	----	----	----

Naprawiamy kopiec. Krok 3: $3 < - > 5$:

3	7	5	10	8	12	19	16	29	11	20	14
---	---	---	----	---	----	----	----	----	----	----	----

Efekt końcowy:

3	7	5	10	8	12	19	16	29	11	20	14
---	---	---	----	---	----	----	----	----	----	----	----

Usuwanie z kopca w implementacji tablicowej - przykład

Dany jest kopiec:

3	7	5	10	8	12	19	16	29	11	20	14
---	---	---	----	---	----	----	----	----	----	----	----

Usuwamy korzeń:

14	7	5	10	8	12	19	16	29	11	20	3
----	---	---	----	---	----	----	----	----	----	----	---

Naprawiamy kopiec. Krok 1: $14 < - > 5$

5	7	14	10	8	12	19	16	29	11	20	3
---	---	----	----	---	----	----	----	----	----	----	---

Naprawiamy kopiec. Krok 2: $14 < - > 12$

5	7	12	10	8	14	19	16	29	11	20	3
---	---	----	----	---	----	----	----	----	----	----	---

Efekt końcowy:

5	7	12	10	8	14	19	16	29	11	20
---	---	----	----	---	----	----	----	----	----	----

Przekształcanie tablicy w kopiec metodą R.Floyda (1964)

Dana jest tablica *data* o rozmiarze *n*.

Poniższy pseudokod przekształca tablicę *data* w kopiec.

```
for (i=indeks ostatniego węzła-nie liścia;  
     i>=0;    i--)  
{  
    odtwórz warunki kopca dla drzewa,  
    którego korzeniem jest data[i] wywołując  
    funkcję MoveDown(data, i, n-1);  
}
```

Przekształcanie tablicy w kopiec metodą R.Floyda (1964)

Funkcja MoveDown

```
void MoveDown(T data[], int first, int last) {  
    int largest = 2* first +1;  
    while (largest <= last) {  
        /* first ma dwa następcy w 2*first+1 oraz 2*first+2;  
        przy czym drugi jest większy od pierwszego */  
        if (largest < last && data[largest] < data[largest+1]) largest ++ ;  
        if (data[first] < data[largest]) {  
            /* jeśli trzeba zamień dziecko z jego rodzicem i przesuń w dół */  
            swap(data[first], data[largest]);  
            first=largest;  
            largest=2*first+1;  
        } else {  
            largest=last+1;  
        }  
    } //while  
}
```

Przekształcanie tablicy w kopiec metodą R.Floyda - przykład

Dana jest tablica A:

2	8	6	1	10	15	3	12	11
---	---	---	---	----	----	---	----	----

Przekształcanie tablicy w kopiec metodą R.Floyda - przykład

Dana jest tablica A:

2	8	6	1	10	15	3	12	11
---	---	---	---	----	----	---	----	----

Krok 1. Ostatni węzeł nie będący liściem: $i = n/2 - 1 = 9/2 - 1 = 3$

0	1	2	3	4	5	6	7	8
2	8	6	1	10	15	3	12	11

Przekształcanie tablicy w kopiec metodą R.Floyda - przykład

Dana jest tablica A:

2	8	6	1	10	15	3	12	11
---	---	---	---	----	----	---	----	----

Krok 1. Ostatni węzeł nie będący liściem: $i = n/2 - 1 = 9/2 - 1 = 3$

0	1	2	3	4	5	6	7	8
2	8	6	1	10	15	3	12	11

Krok 2. Odtwarzamy warunki kopca dla $A[3] = 1$. Zamieniamy 1 z 12

0	1	2	3	4	5	6	7	8
2	8	6	12	10	15	3	1	11

Przekształcanie tablicy w kopiec metodą R.Floyda - przykład

Dana jest tablica A:

2	8	6	1	10	15	3	12	11
---	---	---	---	----	----	---	----	----

Krok 1. Ostatni węzeł nie będący liściem: $i = n/2 - 1 = 9/2 - 1 = 3$

0	1	2	3	4	5	6	7	8
2	8	6	1	10	15	3	12	11

Krok 2. Odtwarzamy warunki kopca dla $A[3] = 1$. Zamieniamy 1 z 12

0	1	2	3	4	5	6	7	8
2	8	6	12	10	15	3	1	11

Krok 3. Kolejny węzeł nie będący liściem: $i = n/2 - 2 = 9/2 - 2 = 2$

Odtwarzamy warunki kopca dla $A[2] = 6$. Zamieniamy 6 z 15

0	1	2	3	4	5	6	7	8
2	8	15	12	10	6	3	1	11

Przekształcanie tablicy w kopiec metodą R.Floyda - przykład

Krok 4. Kolejny węzeł nie będący liściem: $i = n/2 - 3 = 9/2 - 3 = 1$
Odtwarzamy warunki kopca dla $A[1] = 8$. Zamieniamy 8 z 12, a potem 8 z 11.

0	1	2	3	4	5	6	7	8
2	12	15	8	10	6	3	1	11
2	12	15	11	10	6	3	1	8

Przekształcanie tablicy w kopiec metodą R.Floyda - przykład

Krok 4. Kolejny węzeł nie będący liściem: $i = n/2 - 3 = 9/2 - 3 = 1$

Odtwarzamy warunki kopca dla $A[1] = 8$. Zamieniamy 8 z 12, a potem 8 z 11.

0	1	2	3	4	5	6	7	8
2	12	15	8	10	6	3	1	11
2	12	15	11	10	6	3	1	8

Krok 5. Kolejny węzeł nie będący liściem: $i = n/2 - 4 = 9/2 - 4 = 0$

Odtwarzamy warunki kopca dla $A[0] = 2$. Zamieniamy 2 z 15, a potem 2 z 6.

0	1	2	3	4	5	6	7	8
15	12	2	11	10	6	3	1	8
15	12	6	11	10	2	3	1	8

Przekształcanie tablicy w kopiec metodą R.Floyda - przykład

Krok 4. Kolejny węzeł nie będący liściem: $i = n/2 - 3 = 9/2 - 3 = 1$

Odtwarzamy warunki kopca dla $A[1] = 8$. Zamieniamy 8 z 12, a potem 8 z 11.

0	1	2	3	4	5	6	7	8
2	12	15	8	10	6	3	1	11
2	12	15	11	10	6	3	1	8

Krok 5. Kolejny węzeł nie będący liściem: $i = n/2 - 4 = 9/2 - 4 = 0$

Odtwarzamy warunki kopca dla $A[0] = 2$. Zamieniamy 2 z 15, a potem 2 z 6.

0	1	2	3	4	5	6	7	8
15	12	2	11	10	6	3	1	8
15	12	6	11	10	2	3	1	8

Kopiec:

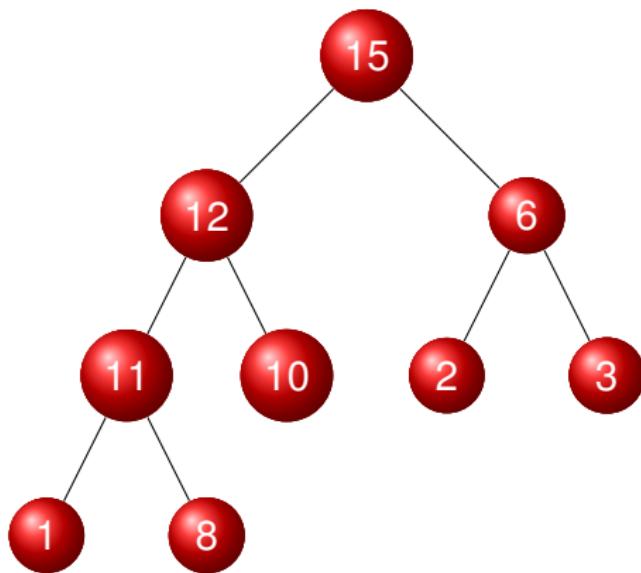
15	12	6	11	10	2	3	1	8
----	----	---	----	----	---	---	---	---

Efekt końcowy:

Kopiec w tablicy:

15	12	6	11	10	2	3	1	8
----	----	---	----	----	---	---	---	---

Kopiec jako drzewo:



Sortowanie przez kopcowanie - implementacja tablicowa

Uwaga! Elementy kopca nie są idealnie uporządkowane (największy znajduje się w korzeniu, a dla każdego węzła spełniony jest warunek, że wszystkie jego następcy nie są większe od wartości danego węzła).

Sortowanie drzewiaste jednowymiarowej tablicy liczb polega na:

- przekształceniu tablicy w kopiec,
- zdjęciu jego korzenia (jako największego elementu),
- przekształceniu tablicy (bez dotychczasowego elementu największego) w nowy kopiec,
- zdjęciu korzenia z otrzymanego kopca (jako kolejnej co do wielkości wartości) itd.

Złożoność czasowa algorytmu: $T(n) = O(n \log_2 n)$

Algorytm sortowania prze kopcowanie

```
void heapsort(T data[], int size) {
    // utworzenie kopca
    for (int i=size/2-1; i>=0; i--)
        MoveDown(data, i, size-1);
    for (int i=size-1; i>=1; i--) {
        // zdjecie korzenia z kopca
        swap(data[ 0 ],data[ i ]);
        // przywracenie性质 kopca
        MoveDown(data, 0, i-1);
    }
}
```

Sortowanie przez Kopcowanie - przykład

Dana jest tablica:

15	12	6	11	10	2	3	1	8
----	----	---	----	----	---	---	---	---

Sortowanie przez Kopcowanie - przykład

Dana jest tablica:

15	12	6	11	10	2	3	1	8
----	----	---	----	----	---	---	---	---

Krok 1: Zamiana korzenia z ostatnim elementem (tj. 15 z 8) i zdjęcie elementu największego (korzenia) z kopca:

8	12	6	11	10	2	3	1	15
---	----	---	----	----	---	---	---	----

Sortowanie przez Kopcowanie - przykład

Dana jest tablica:

15	12	6	11	10	2	3	1	8
----	----	---	----	----	---	---	---	---

Krok 1: Zamiana korzenia z ostatnim elementem (tj. 15 z 8) i zdjęcie elementu największego (korzenia) z kopca:

8	12	6	11	10	2	3	1	15
---	----	---	----	----	---	---	---	----

Krok 2: Przywrócenie drzewu własności kopca. Zamiana 8 z 12, a potem 8 z 11.

12	8	6	11	10	2	3	1	15
12	11	6	8	10	2	3	1	15

Sortowanie przez Kopcowanie - przykład

Dana jest tablica:

15	12	6	11	10	2	3	1	8
----	----	---	----	----	---	---	---	---

Krok 1: Zamiana korzenia z ostatnim elementem (tj. 15 z 8) i zdjęcie elementu największego (korzenia) z kopca:

8	12	6	11	10	2	3	1	15
---	----	---	----	----	---	---	---	----

Krok 2: Przywrócenie drzewu własności kopca. Zamiana 8 z 12, a potem 8 z 11.

12	8	6	11	10	2	3	1	15
12	11	6	8	10	2	3	1	15

Krok 3: Zamiana korzenia z ostatnim elementem (tj. 12 z 1) i zdjęcie elementu największego (korzenia) z kopca:

1	11	6	8	10	2	3	12	15
---	----	---	---	----	---	---	----	----

Sortowanie przez Kopcowanie - przykład, cd.

Krok 4: Przywrócenie drzewu własności kopca. Zamiana 1 z 11, a potem 1 z 10.

11	1	6	8	10	2	3	12	15
11	10	6	8	1	2	3	12	15

Sortowanie przez Kopcowanie - przykład, cd.

Krok 4: Przywrócenie drzewu własności kopca. Zamiana 1 z 11, a potem 1 z 10.

11	1	6	8	10	2	3	12	15
11	10	6	8	1	2	3	12	15

Krok 5: Zamiana korzenia z ostatnim elementem (tj. 11 z 3) i zdjęcie elementu największego (korzenia) z kopca:

3	10	6	8	1	2	11	12	15
---	----	---	---	---	---	----	----	----

Sortowanie przez Kopcowanie - przykład, cd.

Krok 4: Przywrócenie drzewu własności kopca. Zamiana 1 z 11, a potem 1 z 10.

11	1	6	8	10	2	3	12	15
11	10	6	8	1	2	3	12	15

Krok 5: Zamiana korzenia z ostatnim elementem (tj. 11 z 3) i zdjęcie elementu największego (korzenia) z kopca:

3	10	6	8	1	2	11	12	15
---	----	---	---	---	---	----	----	----

Krok 6: Przywrócenie drzewu własności kopca. Zamiana 3 z 10, a potem 3 z 8.

10	3	6	8	1	2	11	12	15
10	8	6	3	1	2	11	12	15

Sortowanie przez Kopcowanie - przykład, cd.

Krok 7: Zamiana korzenia z ostatnim elementem (tj. 10 z 2) i zdjęcie elementu największego (korzenia) z kopca:

2	8	6	3	1	10	11	12	15
---	---	---	---	---	----	----	----	----

Sortowanie przez Kopcowanie - przykład, cd.

Krok 7: Zamiana korzenia z ostatnim elementem (tj. 10 z 2) i zdjęcie elementu największego (korzenia) z kopca:

2	8	6	3	1	10	11	12	15
---	---	---	---	---	----	----	----	----

Krok 8: Przywrócenie drzewu własności kopca. Zamiana 2 z 8, a potem 2 z 3.

8	2	6	3	1	10	11	12	15
8	3	6	2	1	10	11	12	15

itd.

Sortowanie przez Kopcowanie - przykład, cd.

Krok 7: Zamiana korzenia z ostatnim elementem (tj. 10 z 2) i zdjęcie elementu największego (korzenia) z kopca:

2	8	6	3	1	10	11	12	15
---	---	---	---	---	----	----	----	----

Krok 8: Przywrócenie drzewu własności kopca. Zamiana 2 z 8, a potem 2 z 3.

8	2	6	3	1	10	11	12	15
8	3	6	2	1	10	11	12	15

itd.

Końcowa:

1	2	3	6	8	10	11	12	15
---	---	---	---	---	----	----	----	----

Kolejka priorytetowa

Kolejka priorytetowa to struktura danych zawierająca elementy z kluczami, która pozwala na przeprowadzanie dwóch podstawowych operacji:

- wstawiania nowego elementu,
- usuwania elementu o największej (odp. najmniejszej) wartości.

(R. Sedgewick, Algorytmy w C++)

Kolejka priorytetowa

- Kolejka priorytetowa jest strukturą danych, w której o zdaniu elementu z kolejki nie decyduje tylko kolejność umieszczenia elementu w kolejce, ale także priorytet. Pierwszy zostanie obsłużony ten element, który ma największy priorytet.
- Możliwe są dwie metody realizacji kolejki priorytetowej: Kopiec, Lista jednokierunkowa.

Zastosowania kolejki priorytetowej

- Systemy symulacyjne, w których dane kolejki mogą odpowiadać czasom wystąpienia zdarzeń przeznaczonych do chronologicznego przetwarzania.
- Planowanie zadań w systemach komputerowych - dane kolejki mogą oznaczać priorytety wskazujące, którzy użytkownicy mają być obsługiwani w pierwszej kolejności.
- Obliczenia numeryczne, gdzie klucze mogą być wartościami błędów obliczeniowych, oznaczającymi, że największy powinien zostać obsłużony jako pierwszy.



Prace Koła Matematyków Uniwersytetu Pedagogicznego w Krakowie (2015)

Barbara Ciecielska¹, Agnieszka Kowalczyk²

Twierdzenie Perrona–Frobeniusa i jego zastosowanie w algorytmie Page Rank

Streszczenie. Jednym z ciekawszych i znajdujących wiele zastosowań twierdzeń z algebry liniowej jest twierdzenie Perrona–Frobeniusa. Twierdzenie to jest wspólnym rezultatem prac Oskara Perrona z 1907 i Georga Frobeniusa z 1912. Dzięki niemu wiadomo, iż największa wartość własna rzeczywistej nieujemnej kwadratowej macierzy jest rzeczywista i ma krotność 1. Co więcej, wektor własny korespondujący z tą wartością własną jest ściśle dodatni. Twierdzenie to znajduje zastosowanie w teorii prawdopodobieństwa, ekonomii, demografii, rankingach, a także (dzięki idei Larry’ego Page’a i Sergey’ego Brina z 1996 roku) w silnikach wyszukiwarek internetowych. Głównym celem naszego artykułu jest przedstawienie twierdzenia Perrona–Frobeniusa wraz z zarysem jednego z najpopularniejszych jego dowodów (korzystającego z twierdzenia Brouwera o punkcie stałym) oraz zaprezentowanie jego różnorakich i zaskakujących zastosowań, między innymi w algorytmie Page Rank używanym obecnie przez wyszukiwarkę Google.

Abstract. The Perron–Frobenius theorem, which was firstly proved by Oskar Perron in 1907 and later extended by Georg Frobenius in 1912, asserts that a real nonnegative square matrix has a unique largest real eigenvalue and that the eigenvector corresponding to it has strictly positive components and that this eigenvector is stochastic. This theorem has a wide variety of applications: from probability theory, through economics, demography and rankings to (according to Larry Page and Sergey Brin’s idea in 1996) Internet search engines. The main aim is to present this theorem with one of its most popular proofs (using the Brouwer fixed point theorem) and to explain the idea of several of its applications.

AMS (2010) Subject Classification: 15A18, 15A42.

Slowa kluczowe: tw. Perrona–Frobeniusa, algorytm PageRank, model Lesliego, rankingi, Power Control Problem.

1. Wstęp

1.1. Wstęp historyczny

Twierdzenie Perrona–Frobeniusa jest twierdzeniem z zakresu algebry liniowej noszącym nazwiska dwóch niemieckich matematyków. Obaj zajmowali się głównie matematyką teoretyczną.

Oskar Perron urodził się 7 maja 1880 roku w Frankenthal i zmarł 22 lutego 1975 roku w Monachium. Był profesorem Uniwersytetu w Heidelbergu i Uniwersytetu Ludwika Maksymiliana w Monachium. Zajmował się głównie równaniami różniczkowymi zwyczajnymi i cząstkowymi, a także mechaniką nieba i teorią macierzy. To właśnie on w roku 1907 opublikował i udowodnił pierwotną wersję twierdzenia w czasopiśmie „Mathematische Annalen”, w artykule noszącym tytuł „Zur Theorie der Matrices” (niem. „O teorii macierzy”).

Z kolei Ferdinand Georg Frobenius urodził się 26 października 1849 roku w Charlottenburgu (obecnie dzielnica Berlina) i zmarł 3 sierpnia 1917 tamże. Był związany głównie z Uniwersytetem Humboldta w Berlinie i Politechniką Federalną w Zurychu. Do jego głównych matematycznych zainteresowań należały: teoria grup, teoria algebr, teoria liczb i równania różniczkowe. Zainteresował się również twierdzeniem udowodnionym przez Perrona i na przestrzeni lat 1908–1912 opublikował trzy prace z nim związane. Ostatnia, pochodząca z roku 1912, nosi tytuł „Über Matrizen aus nicht negativen Elementen” (niem. „O macierzach posiadających nieujemne wyrazy”) i zawiera ostateczną, obecną wersję twierdzenia Perrona–Frobeniusa.

Zarówno Perron, jak i Frobenius zajmowali się czystą matematyką i nie interesowali się zbytnio jej zastosowaniami. Frobenius uważało wręcz, iż zastosowania matematyki powinny skupiać zainteresowanie technicznych uczelni. Żaden z nich nie przewidział, jak szeroki zakres zastosowań będzie mieć ich twierdzenie. W XX wieku twierdzenie to zostało wykorzystane w modelowaniu ekonomicznym (m. in. w: równaniu wymiany, modelu Leontiefa czy też liniowym modelu produkcji), w modelowaniu biologicznym, a także — używanym praktycznie przez każdego internautę — algorytmie Page Rank.

1.2. Wstęp teoretyczny

TWIERDZENIE 1 (TWIERDZENIE BROUWERA O PUNKCIE STAŁYM)

Jeżeli X jest niepustym, domkniętym, ograniczonym i wypukłym podzbiorem \mathbb{R}^n , a $F: X \rightarrow X$ odwzorowaniem ciągłym, to istnieje takie $x \in X$, że $F(x) = x$.

DEFINICJA 2 (WARTOŚĆ WŁASNA, WEKTOR WŁASNY)

Wektorem własnym kwadratowej macierzy $A \in \mathbb{R}^{n \times n}$ nazywamy taki niezerowy wektor v , dla którego zachodzi $Av = \lambda v$ dla pewnego $\lambda \in \mathbb{R}$.

Skalar λ nazywamy wartością własną macierzy A odpowiadającą wektorowi własnemu v .

DEFINICJA 3 (WIDMO, PROMIĘŃ SPEKTRALNY MACIERZY)

Widmem macierzy $A \in \mathbb{R}^{n \times n}$ nazywamy zbiór

$$\sigma(A) = \{\lambda \in \mathbb{R} : \det(A - \lambda I) = 0\}.$$

Promieniem spektralnym macierzy A nazywamy liczbę

$$\max \{|\lambda| : \lambda \in \sigma(A)\}.$$

DEFINICJA 4 (MACIERZ REDUKOWALNA)

Kwadratową macierz $A \in \mathbb{R}^{n \times n}$ nazywamy redukowalną, jeżeli za pomocą permutacji odpowiednich wierszy i kolumn możemy ją sprowadzić do postaci górnej trójkątnej. To znaczy, że macierz jest redukowalna, jeżeli istnieje taka macierz permutacji P , że zachodzi:

$$P^T A P = \begin{bmatrix} B & C \\ 0 & D \end{bmatrix},$$

gdzie B i D to macierze kwadratowe, a 0 to macierz zerowa.

DEFINICJA 5 (MACIERZ NIEREDUKOWALNA)

Macierz A nazywamy nieredukowalną, jeżeli nie jest redukowalna.

Dysponujemy również równoważną definicją:

Mówimy, że macierz $A \in \mathbb{R}^{n \times n}$ jest nieredukowalna, gdy dla każdej pary $(i, j) \in \mathbb{N} \times \mathbb{N}$ istnieje takie całkowite $m > 0$, że $(A^m)_{ij} > 0$, gdzie (A^m) to m -ta potęga macierzy A .

PRZYKŁAD 6

Macierz

$$\begin{bmatrix} 5 & 7 & 0 \\ 6 & 0 & 0 \\ 1 & 0 & 2 \end{bmatrix}$$

jest redukowalna. Natomiast macierz

$$\begin{bmatrix} 1 & 2 & 1 \\ 2 & 1 & 6 \\ 0 & 1 & 7 \end{bmatrix}$$

jest nieredukowalna.

DEFINICJA 7 (MACIERZ PRYMTYWNA)

Macierz $A \in \mathbb{R}^{n \times n}$ o wyrazach nieujemnych nazywamy prymitywną, jeżeli istnieje taka liczba naturalna b , że zachodzi $(A^b)_{ij} > 0$.

UWAGA 8

Macierz prymitywna jest nieredukowalna.

DEFINICJA 9 (WEKTOR PRAWDOPODOBIĘSTWA/STOCHASTYCZNY)

Wektorem prawdopodobieństwa nazywamy taki wektor $v \in \mathbb{R}^n$, którego wszystkie współrzędne są nieujemne i sumują się do jedności.

PRZYKŁAD 10

Wektorami prawdopodobieństwa są wektory $x = (1, 0, 0, 0)$, $y = (\frac{1}{2}, \frac{1}{4}, 0, 0, \frac{1}{4})$, ale nie są nimi $z = (1, 2, 0, 0)$, $w = (-\frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2})$.

DEFINICJA 11 (MACIERZ MARKOWA/STOCHASTYCZNA/PROBABILISTYCZNA)

Kwadratową macierz $S \in \mathbb{R}^{n \times n}$ nazywamy prawą (lewą) macierzą Markowa, jeżeli wszystkie jej elementy to nieujemne liczby rzeczywiste, a elementy w każdym jej wierszu (kolumnie) sumują się do jedności.¹

PRZYKŁAD 12

Macierz

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \end{bmatrix}$$

jest prawą macierzą Markowa.

DEFINICJA 13 (ŁAŃCUCH MARKOWA)

Łańcuchem Markowa nazywamy taki ciąg wektorów prawdopodobieństwa x_0, x_1, x_2, \dots , że zachodzi $x_{k+1} = Mx_k$ dla pewnej lewej macierzy Markowa M .

UWAGA 14

Łańcuchem Markowa jest więc ciąg zdarzeń, w którym prawdopodobieństwo każdego zdarzenia zależy jedynie od stanu poprzedniego. Łańcuch Markowa to dyskretny proces Markowa, czyli proces stochastyczny², który spełnia tzw. własność Markowa:

$$\mathbb{P}(X_{n+1} = x_{n+1} | X_0 = x_0, X_1 = x_1, \dots, X_n = x_n) = \mathbb{P}(X_{n+1} = x_{n+1} | X_n = x_n),$$

gdzie $X_0, X_1, X_2, \dots, X_n$ to ciąg zmiennych losowych.

TWIERDZENIE 15

Jeżeli M jest macierzą Markowa, to istnieje taki niezerowy wektor v , dla którego zachodzi $Mv = v$. To oznacza, że każda taka macierz posiada wartością własną równą 1.

DEFINICJA 16

Powyższy wektor v nazywamy wektorem stacjonarnym.

PRZYKŁAD 17

W Krakowie kibice piłki nożnej³ solidaryzują się z Cracovią bądź z Wisłą Kraków (dla uproszczenia pomijamy pozostałe drużyny piłkarskie). Niech $x_0 = (0, 4; 0, 6)$ (tzn. że 40% kibiców solidaryzuje się z Cracovią, a 60% z drużyną z drugiej strony Błoń). Każdego roku 10% kibiców Wisły zmienia zdanie i zaczyna kibicować Cracovii, pozostali pozostają wierni swojej drużynie. Analogicznie, każdego roku 5% kibiców opuszcza Cracovię i wybiera Wisłę, a pozostali zostają. Oznaczmy $x_k = (c_k; w_k)$ jako wektor przedstawiający preferencje kibiców w k -tym roku. Zapiszmy ten problem w postaci macierzowej:

$$\begin{bmatrix} c_{k+1} \\ w_{k+1} \end{bmatrix} = \begin{bmatrix} 0, 95 & 0, 1 \\ 0, 05 & 0, 9 \end{bmatrix} \begin{bmatrix} c_k \\ w_k \end{bmatrix}$$

¹To oznacza, że wiersze (kolumny) są wektorami prawdopodobieństwa.

²Procesem stochastycznym nazywamy rodzinę zmiennych losowych indeksowaną parametrem t interpretowanym jako czas

³Ten przykład jest uproszczeniem sytuacji panującej w Mieście Królów Polskich i nie ma na celu nikogo urazić.

Zachodzą następujące równości:

$$c_{k+1} = 0,95c_k + 0,1w_k,$$

$$w_{k+1} = 0,9w_k + 0,05c_k.$$

Przykładowo wyliczmy: $x_1 = (0, 44; 0, 56)$, $x_2 = (0, 474; 0, 526)$. Natomiast dla dużych k wektor x_k zmierza do $(\frac{2}{3}, \frac{1}{3})$ – jest to wektor stacjonarny.

DEFINICJA 18 (MACIERZ SĄSIEDZTWA)

Kwadratową macierz $A \in \mathbb{R}^{n \times n}$ nazywamy macierzą sąsiedztwa, gdy $a_{ij} = 0$ lub $a_{ij} = 1$. Macierz ta opisuje graf o n wierzchołkach.

DEFINICJA 19 (GRAF STOWARZYSZONY Z MACIERZĄ SĄSIEDZTWA)

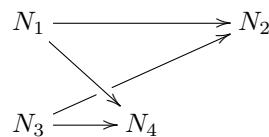
Graf $\mathcal{G}(A)$ dla danej macierzy sąsiedztwa $A \in \mathbb{R}^{n \times n}$ jest definiowany jako graf z n wierzchołkami (N_1, \dots, N_n) , dla którego istnieje krawędź skierowana z N_j do N_i wtedy i tylko wtedy, gdy $a_{ij} \neq 0$.

PRZYKŁAD 20

Dla macierzy:

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix}$$

mamy następujący graf:

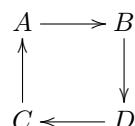


DEFINICJA 21 (GRAF SILNIE SPÓJNY)

Graf jest silnie spójny, jeżeli między każdą parą wierzchołków istnieje ścieżka (niekoniecznie krawędź skierowana).

PRZYKŁAD 22

Graf



jest silnie spójny. Graf z przykładu 20 nie jest silnie spójny.

UWAGA 23

Macierz $A \in \mathbb{R}^{n \times n}$ jest nieredukowalna wtedy i tylko wtedy, gdy graf $\mathcal{G}(A)$ jest silnie spójny.

2. Twierdzenie Perrona–Frobeniusa

TWIERDZENIE 24 (TWIERDZENIE PERRONA–FROBENIUSA)

Jeżeli wszystkie wyrazy niereductowalnej macierzy $A \in \mathbb{R}^{n \times n}$ są nieujemne, to istnieje dokładnie jedna taka rzeczywista dodatnia wartość własna tej macierzy $\lambda > 0$ (o krotności algebraicznej 1), że dla wszystkich innych wartości własnych μ tej macierzy zachodzi $|\mu| < \lambda$. Ponadto istnieje taki wektor własny stwarzyszony z λ , że wszystkie jego współczynniki są dodatnie.

Dowód. Rozważmy zbiór

$$\mathcal{C} = \{(x_1, \dots, x_n) \in \mathbb{R}^n : x_i \geq 0\}$$

oraz standardowy sympleks:

$$\mathcal{S} = \left\{ (x_1, \dots, x_n) \in \mathbb{R}^n : x_i \geq 0, \sum_{i=1}^n x_i = 1 \right\} \subset \mathcal{C}.$$

Na sympleksie \mathcal{S} definiujemy następujące odwzorowanie: $T(x) = \frac{Ax}{(Ax) \cdot (1, 1, \dots, 1)}$. Zwróćmy uwagę, że wyrażenie $(Ax) \cdot (1, 1, \dots, 1)$ jest różne od zera na \mathcal{S} , bo macierz A jest niereductowalna i o nieujemnych wyrazach (żadna jej kolumna nie jest zerowa). Widzimy zatem, że odwzorowanie T jest poprawnie określone i ciągłe jako iloraz odwzorowań ciągłych. Łatwo również zauważyc, że $T(\mathcal{S}) \subset \mathcal{S}$, ponieważ:

- wszystkie współczynniki macierzy A i wektora należącego do sympleksu x są nieujemne, a co za tym idzie – także wektora Ax , zatem również wyrażenia $\frac{Ax}{(Ax) \cdot (1, 1, \dots, 1)}$,
- suma współrzędnych wektora $T(x)$ jest równa 1, ponieważ odwzorowanie możemy zapisać następująco: $T(x) = \left(\frac{\sum_{i=1}^n A_{i,1}x_i}{\sum_{i=1}^n \sum_{j=1}^n A_{i,j}x_i}, \dots, \frac{\sum_{i=1}^n A_{i,n}x_i}{\sum_{i=1}^n \sum_{j=1}^n A_{i,j}x_i} \right)$, a wtedy łatwo widać, że $\sum_{j=1}^n \frac{\sum_{i=1}^n A_{i,j}x_i}{\sum_{i=1}^n \sum_{j=1}^n A_{i,j}x_i} = 1$.

\mathcal{S} jest zbiorem niepustym, domkniętym, ograniczonym i wypukłym, a T jest odwzorowaniem ciągłym. Spełnione są więc założenia twierdzenia Brouwera o punkcie stałym. Istnieje zatem taki punkt x , że $T(x) = x$, czyli $\frac{Ax}{(Ax) \cdot (1, 1, \dots, 1)} = x$, a więc:

$$Ax = ((Ax) \cdot (1, 1, \dots, 1))x.$$

Pokażemy, że każdy punkt stały tego odwzorowania leży wewnątrz sympleksu – czyli wszystkie jego współrzędne są dodatnie. Założmy dla dowodu nie wprost, że tak nie jest i x należy do pewnej ściany $(n-2)$ -wymiarowej S_j leżącej naprzeciw wierzchołka e_j . Przede wszystkim zauważmy, że punkt stały odwzorowania T jest też punktem stałym każdego odwzorowania T^N . Ale wiemy, że macierz

A jest nieredukowalna, więc istnieje takie N , że $A_{i,j}^N \neq 0$, zatem zachodzi również $T^N(S_j) \not\subset S_j$. To pokazuje, że punkt stały nie może leżeć w wnętrzu tegoż sympleksu. Analogicznie traktujemy $(n-k)$ -wymiarowe krawędzie dla $k \in \{3, \dots, n-1\}$, a także w końcu wierzchołki: e_i nie może być punktem stałym, gdyż istnieją N i j takie, że $A_{i,j}^N \neq 0$. Stąd otrzymujemy sprzeczność: jeżeli $T(x) = x$, to $x \notin \partial\mathcal{S}$. Wiemy więc z tego, że wszystkie współrzędne wszystkich punktów stałych naszego odwzorowania są dodatnie.

Największą dodatnią wartością własną, której stwarzyszyliśmy wektorem własnym jest x , otrzymujemy następująco:

$$\lambda_{max} = \max\{(Ax) \cdot (1, 1, \dots, 1), x = T(x)\}.$$

Określmy teraz odwzorowanie $L: \mathcal{S} \rightarrow \mathbb{R}$ poniższym wzorem:

$$L(z) = \max\{s \in \mathbb{R}_+: sz \leq Az\},$$

gdzie przez relację \leq rozumiemy porównywanie współrzędnych. Określmy ponadto macierz $P := (I + A)^q$, gdzie $q \in \mathbb{N}$ jest dobrane tak, że P jest macierzą dodatnią. Z określenia L wiemy, że $L(rz) = L(z)$ dla każdego niezerowego r . Jeżeli $z < y$ (porównujemy współrzędne wektorów), to $Pz < Py$ oraz $PA = AP$. Czyli dla $sz \leq Az$ zachodzi $sPz \leq PAz = APz$, a to implikuje $L(z) \leq L(Pz)$. Wiemy również, że obraz \mathcal{S} przez P jest zwarty (jako obraz zwartego zbioru przez odwzorowanie ciągłe) i wszystkie elementy P są dodatnie. Funkcja L jest ciągła na $P(\mathcal{S})$, czyli osiąga maksimum na $P(\mathcal{S})$ – powiedzmy L_{max} . Ponieważ zachodzi $L(z) \leq L(Pz)$ i wiemy, że $L(z) < L(Pz)$ (chyba że z jest wektorem własnym A), wnioskujemy, że L_{max} jest osiągalne w wektorze własnym x , a L_{max} to wartość własna.

Weźmy teraz wektor własny w macierzy A i stwarzyszoną z nim wartość własną μ . Jako $|w_j|$ oznaczamy j -tą współrzędną wektora w . Przeprowadzamy następujące proste oszacowanie:

$$|\mu||w|_i = |\mu w_i| = \left| \sum_j A_{ij} w_j \right| \leq \sum_j |A_{ij}| |w_j| = \sum_j A_{ij} |w_j| = (A|w|)_i.$$

Pokrótce możemy zapisać, że $|\mu||w| \leq A|w|$, czyli – wracając do określenia L – otrzymujemy, iż $\mu \leq L(|y|) \leq L_{max}$. Ponieważ zachodzi $L_{max} = \lambda_{max}$, to otrzymujemy $|\mu| \leq \lambda_{max}$.

Teraz udowodnimy, że λ_{max} ma krotnością algebraiczną 1. Zauważmy najpierw, że jeżeli pochodna wielomianu charakterystycznego macierzy A (w punkcie λ_{max}) jest różna od zera, to oczywiście λ_{max} nie jest pierwiastkiem dwukrotnym tego wielomianu, czyli ma krotnością algebraiczną również 1. Rozważmy więc A (naszą kwadratową macierz założenia) i C – diagonalną macierz takich samych wymiarów, co A , z wartościami $\lambda_1, \dots, \lambda_n$ na diagonali. Korzystając z metody Laplace'a liczenia wyznacznika macierzy łatwo zobaczyć, że $\frac{\partial}{\partial \lambda_i} \det(C - A) = \det(C_i - A_i)$, gdzie A_i oznacza macierz utworzoną przez wykreślenie i -tej kolumny i i -tego wiersza, tak samo C_i . Kładąc $\lambda_i = \lambda$ i korzystając z reguły Leibniza mamy, że

$$\frac{\partial}{\partial \lambda_i} \det(\lambda I - A) = \sum_i \det(\lambda I_i - A_i).$$

Ponieważ każda z macierzy $\lambda_{max}I_i - A_i$ ma dodatni wyznacznik, to pochodna wielomianu charakterystycznego macierzy A po λ_{max} (w punkcie λ_{max}) jest większa od zera, czyli λ_{max} ma krotność algebraiczną 1.

Na koniec zauważmy jeszcze, że mając $|\mu| \leq \lambda$ i krotność algebraiczną 1 wartościowej λ (czyli $\mu \neq \lambda$) otrzymujemy $|\mu| < \lambda$, co już kończy dowód.

UWAGA 25

Od tego momentu λ oznaczać będzie największą rzeczywistą wartość własną dla aktualnie rozważanej macierzy.

UWAGA 26

Jeżeli macierz A spełnia założenia twierdzenia Perrona–Frobeniusa, to istnieje dokładnie jeden stochastyczny wektor własny $x = (x_1, \dots, x_n)$ tej macierzy stowarzyszony z wartością własną λ .

UWAGA 27

Zauważmy, że jeśli macierz A spełnia założenia twierdzenia Perrona–Frobeniusa i jeśli dodatkowo A jest macierzą Markowa, to wektor x z tezy twierdzenia jest wektorem stacjonarnym procesu Markowa. Dzięki temu twierdzeniu otrzymujemy dodatkowo jego jedynosć.

WNIOSEK 28

Jeśli macierz $A \in \mathbb{R}^{n \times n}$ jest stochastyczna i nieredukowalna, to promień spektralny tej macierzy wynosi 1 i istnieje dokładnie jeden dodatni stochastyczny wektor własny $x = (x_1, \dots, x_n)$. Ponadto, $x = \frac{1}{n} \lim_{k \rightarrow \infty} A^k e$, gdzie $e = (1, \dots, 1)$.

3. Zastosowania

3.1. Rankingi

Metod i podstaw do tworzenia rankingów są krocie, tak samo jak wyników tych rankingów utworzonych z różnych metod. Są przydatne nie tylko w różnego rodzaju zawodach, lecz także w dyskusjach i sytuacjach, gdy ciężko o bezpośrednie porównanie. Na przykład: gracz A wygrał z graczem B i C, gracze B i C zremisowali, w związku z czym gracze B i C mają tyle samo punktów, ale któryż z nich lepiej się zaprezentował. Z kolei jeśli gracz A wygrywa z B, gracz B z C, C wygrywa z A, to nie ma da się ustalić porządku. Zachodzą również sytuacje, gdy gracze rozegrali nierówne ilości walk, co powoduje trudności w ustaleniu pozycji rankingowej i sposobie oceny.

Jednakże, niezależnie od metody i celu sporządzania rankingu, najpierw musimy mieć elementy, które chcemy uporządkować. My posłużymy się pięcioma dowcipami matematycznymi, z użyciem których zostało przeprowadzone pewne badanie statystyczne. Oto one:

1. Idzie Jezus przez pustynię z Apostołami i naucza:
– Życie jest jak $y = x^2 + 6x - 9$.
– Ej, ale o co mu chodzi? – pyta Tomasz Jana.

–Nie wiem, to chyba jakaś parabola.

2. Do baru *The Legends* wchodzi nieskończenie wiele matematyków.
Pierwszy zamawia piwo.
Drugi zamawia pół piwa.
Trzeci zamawia kwartę piwa.
Barmanka kładzie na barze 2 piwa i mówi:
–Panowie, znajcie swoje granice!
3. Olimpiada odbywała się na kwadratowym stadionie. Niestety poprzedniemu złotemu medaliście w biegu na 400 m zawody nie poszły i zajął dopiero 7 miejsce.
–Co się stało? – pyta dziennikarz.
–Nie jestem w formie kwadratowej.
4. Trzech ludzi leciało balonem nad pustynią, ale stracili orientację i postanowili dowiedzieć się, gdzie się znajdują. Obniżyli lot i widząc na dole człowieka zapytali:
– Czy może nam pan powiedzieć, gdzie jesteśmy?
– W balonie – odpowiedział człowiek na ziemi.
– Pan zapewne jest matematykiem.
– A z czego panowie wnioskują?
– Podał pan odpowiedź prawdziwą, precyzyjną i zupełnie bezużyteczną.
– A panowie zapewne są inżynierami.
– A skąd taki wniosek?
– Po pierwsze: nie macie pojęcia, gdzie jesteście i jak się tam znaleźć; po drugie: prosicie o pomoc matematyka, a po uzyskaniu odpowiedzi dalej nic nie wiecie i winicie za to matematyka.
5. –Jaki jest obraz piwa?
–Imbir.

Badanie statystyczne dotyczące powyższych dowcipów zostało przeprowadzone na pięciu osobach, dysponujących wystarczającą wiedzą matematyczną, by zrozumieć powyższe żarty. Każdy miał do wykonania 2 zadania. Pierwsze polegało na ocenie 4 „pojedynków dowcipów”, tzn. zdecydowaniu, który z dwóch dowcipów jest lepszy (co pytana osoba rozumie pod tym pojęciem, jest kwestią indywidualną). Pojedynek był wyznaczony dla każdej osoby w sposób losowy, ale tak, aby każde dwa dowcipy rozegrały dwie walki. Ostatecznie każdy dowcip walczył 4 razy, więc zostało przeprowadzone w sumie 20 pojedynków. Drugie zadanie polegało na uszeregowaniu dowcipów od najlepszego do najgorszego, przy czym najlepszy dostawał 5 punktów, drugi z kolei – 4, trzeci – 3, czwarty – 2, a ostatni tylko 1. Wyniki zostały zebrane w poniższych tabelach.

Wyniki pojedynków z zadania 1 (wygrane i -tego gracza są wypisane w i -tym wierszu) to:

	1	2	3	4	5	Σ
1	X	0	0	0	1	1
2	2	X	0	0	1	3
3	2	1	X	0	1	4
4	2	2	2	X	1	7
5	1	1	1	1	X	4

Ranking według sumy uzyskanych punktów w zadaniu 1 wygląda tak:

<i>i</i>	v_i	Pozycja rankingowa 1
1	1	5
2	3	4
3	4	2
4	7	1
5	4	2

Z kolei wyniki drugiego zadania i ich ranking wyglądają następująco:

<i>i</i>	v_i	Pozycja rankingowa 2
1	8	5
2	13	4
3	18	1
4	16	3
5	17	2

Na podstawie tabeli z zadania 1 tworzymy macierz A . I tu już pojawia się pierwsza trudność, ponieważ istnieje wiele możliwości określenia tej macierzy. Oto kilka popularnych sposobów używanych na określenie wyrazu a_{ij} :

- a_{ij} – ilość sytuacji, w których dowcip i pokonał dowcip j ,
- $a_{ij} = \frac{S_{ij}}{S_{ij} + S_{ji}}$, gdzie S_{ij} – punkty, które i zdobyło w pojedynku przeciwko j ,
- $a_{ij} = \frac{S_{ij} + 1}{S_{ij} + S_{ji} + 2}$, gdzie S_{ij} – punkty, które i zdobyło w pojedynku przeciwko j ,
- $a_{ij} = h\left(\frac{S_{ij}}{S_{ij} + S_{ji}}\right)$, gdzie S_{ij} – punkty, które i zdobyło w pojedynku przeciwko j , $h(x) = \frac{1}{2} + \frac{1}{2} \operatorname{sgn}(x - \frac{1}{2})\sqrt{|2x - 1|}$.

My w dalszych rozważaniach będziemy korzystać z pierwszego sposobu. Ponieważ ta metoda rankingowa ma na celu skorzystanie z twierdzenia Perrona–Frobeniusa, musimy zadbać, żeby założenia były spełnione. W związku z tym dopisujemy jedynki na diagonali (nie zmieni to ogólności rankingu), aby macierz A była niereductowalna. Otrzymujemy macierz:

$$A = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 \\ 2 & 1 & 0 & 0 & 1 \\ 2 & 1 & 1 & 0 & 1 \\ 2 & 2 & 2 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Największa wartość własna λ macierzy A wynosi $\lambda \approx 4,02$, natomiast stochastyczny wektor własny v stwarzyszony z λ wynosi w przybliżeniu:

$$v \approx (0,17; 0,28; 0,37; 0,71; 0,51).$$

Wartości te pomogą nam w stworzeniu trzeciego rankingu: im wyższa wartość i -tej współrzędnej wektora własnego, tym wyżej i -ty dowcip będzie w naszym zestawieniu. Rankingi jeden i dwa będą wyznaczone za pomocą zadań 1 i 2 z badania statystycznego. W pierwszym rankingu dowcip otrzyma tyle punktów, ile wygrał pojedyneków. Natomiast w drugi ranking polegał będzie na zsumowaniu punktów uzyskanych przez poszczególne dowcipy podczas 2 zadania wykonywanego podczas ankiety. Każda z tych metod wydaje się rozsądny sposobem wyboru najlepszego dowcipu.

Wyniki uzyskane w zadaniu pierwszym przez pryzmat twierdzenia Perrona–Frobeniusa i trzeci ranking przedstawiają się jak niżej:

i	v_i	Pozycja rankingowa 3
1	0,17	5
2	0,28	4
3	0,37	3
4	0,71	1
5	0,51	2

Porównanie jest przedstawione w poniższej tabeli:

i	Pozycja rankingowa 1	Pozycja rankingowa 2	Pozycja rankingowa 3
1	5	5	5
2	4	4	4
3	2	1	3
4	1	3	1
5	2	2	2

Jak widać, różnice w uzyskanej pozycji nie są znaczne, dotyczą tylko pierwszych trzech miejsc, a różnice wskaźników v_i są w paru przypadkach niewielkie. Warto jednak zauważyć, że próbka statystyczna była mała – wynosiła tylko 10 osób. Widzimy jak łatwo manipulować ostatecznymi wynikami rankingu – wystarczy posłużyć się innym systemem liczenia.

Warto wspomnieć jeszcze o jednej modyfikacji macierzy pojedyneków A . Rozważmy macierz $R = \alpha A + (1 - \alpha)E$ dla $\alpha \in (0, 1)$ oraz:

$$E = \begin{bmatrix} v_1 & v_1 & v_1 & v_1 & v_1 \\ v_2 & v_2 & v_2 & v_2 & v_2 \\ v_3 & v_3 & v_3 & v_3 & v_3 \\ v_4 & v_4 & v_4 & v_4 & v_4 \\ v_5 & v_5 & v_5 & v_5 & v_5 \end{bmatrix},$$

gdzie $v_1 + v_2 + v_3 + v_4 + v_5 = 1$. Liczba v_i ma opisywać „jakość i -tego dowcipu. Wystarczy rzucić okiem na tą modyfikację, aby zobaczyć, jak bardzo – w zależności od dobrania wag v_i i definicji „jakości” – różniłyby się wyniki rankingu...

3.2. Power Control Problems

Następnym zastosowaniem tytułowego twierdzenia są tzw. problemy kontroli siły. Aby lepiej zobrazować to zagadnienie, wyobraźmy sobie, że jesteśmy w kawiarni ze znajomymi. Chcemy rozmawiać na komfortowym poziomie głośności. Jednakże hałas dochodzący z naszego stolika powoduje, że ludzie przy innych stolikach prowadzą swoje rozmowy głośniej, aby lepiej siebie rozumieć. To z kolei powoduje, że poziom głośności przy naszym stoliku wzrasta – i tak dalej... (oczywiście pomijamy hałasy postronne). Chcemy zatem znaleźć takie optymalne dodatnie poziomy głośności P_i dla każdego stolika, aby wszystkie osoby w kawiarni mogły się bez problemu porozumiewać w obrębie stolika, przy którym siedzą.

Matematycznie rzecz ujmując – chcemy, aby zachodziły poniższe nierówności:

$$\frac{P_i}{\sum_{i \neq j} G_{i,j} P_j} \geq \gamma_i, \quad i = 1, 2, \dots, m,$$

gdzie danymi są:

- liczba stolików ($m > 0$),
- akceptowalny stosunek głośności rozmów stolika i do hałasu z innych stolików, który dochodzi do stolika i ($\gamma_i > 0$),
- dodatni wskaźnik $G_{i,j}$ osłabienia dźwięku dochodzącego ze stolika j do stolika i (np. wartość wskaźnika $G_{i,j}$ wynosząca $\frac{1}{3}$ oznacza, że przy poziomie głośności a stolika j , stolik i będzie słyszał rozmowę stolika j na poziomie głośności $\frac{2}{3}a$).

Chcemy, aby stosunek głośności był większy bądź równy akceptowalnemu stonkowi głośności. Jeżeli ów stosunek jest mniejszy niż wartość γ_i , oznacza to, że ludzie siedzący przy stoliku i nie mogą siebie usłyszeć pośród wszystkich odgłosów w kawiarni.

Po przemnożeniu przez wartość mianownika, dostajemy:

$$\underbrace{\begin{bmatrix} \gamma_1 & 0 & \dots & 0 \\ 0 & \gamma_2 & \dots & 0 \\ \vdots & \vdots & \ddots & 0 \\ 0 & 0 & \dots & \gamma_m \end{bmatrix}}_A \underbrace{\begin{bmatrix} 0 & G_{1,2} & \dots & G_{1,m} \\ G_{2,1} & 0 & \dots & G_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ G_{m,1} & G_{m,2} & \dots & 0 \end{bmatrix}}_P \underbrace{\begin{bmatrix} P_1 \\ P_2 \\ \vdots \\ P_m \end{bmatrix}}_P \leq \underbrace{\begin{bmatrix} P_1 \\ P_2 \\ \vdots \\ P_m \end{bmatrix}}_P$$

Relację \leq rozumiemy jako porównywanie odpowiednich współrzędnych wektorów. Oznaczmy przez A wymnożone przez siebie dwie pierwsze macierze. Macierz ta jest prymitywna (wystarczy $k = 2$), a więc redukowalna. Wszystkie jej elementy

są nieujemne. W tym właśnie momencie przychodzi nam z pomocą twierdzenie Perrona–Frobeniusa. Mówiąc nam ono, że istnieją taki wektor P_λ o wyrazach dodatnich i taki dodatni skalar λ , że $AP_\lambda = \lambda P_\lambda$. Zauważmy, że nierówność $AP \leq P$ zajdzie dla $P = P_\lambda$ wtedy i tylko wtedy, gdy λ będzie mniejsze bądź równe jedności.

Musimy jeszcze tylko znaleźć akceptowalne ograniczenie dla γ_i – uzyskamy je przy pomocy twierdzenia Siemiona Gerszgorina.

TWIERDZENIE 29 (TWIERDZENIE GERSZGORINA)

Niech $A \in \mathbb{R}^{n \times n}$ będzie macierzą o wyrazach a_{ij} . Niech $R_i = \sum_{j \neq i} |a_{ij}|$ dla $i = 1, \dots, n$. Innymi słowy, R_i to suma modułów wyrazów danego wiersza, które nie leżą na diagonali. Wtedy każda wartość własna λ_i macierzy A zawiera się w sumie każda $D(a_{ii}, R_i) \subset \mathbb{C}$ o środkach w a_{ii} i o promieniach R_i . Tzn. zachodzi: $|\lambda_i - a_{ii}| \leq R_i$.

Aby lepiej zobrazować powyższe twierdzenie pomocnicze, weźmy na przykład wartości własne macierzy $A = \begin{bmatrix} 3 & 2 \\ 4 & 1 \end{bmatrix}$, którymi są -1 i 5 . Leżą one w końcach $D(3, 2)$ oraz $D(1, 4)$.

Stosując to twierdzenie do rozwiązywania naszego problemu, dla $i = 1, \dots, m$ otrzymujemy:

$$|\lambda_i - a_{ii}| = |\lambda_i| \leq \sum_{i \neq j} R_i = \sum_{i \neq j} |a_{ij}| = \gamma_i \sum_{i \neq j} G_{i,j}.$$

Chcemy, by dla każdego i zachodziło $|\lambda_i| \leq 1$. Naturalne jest więc, aby ustalić akceptowalne ograniczenie dla γ_i jako: $\gamma_i \leq \frac{1}{\sum_{i \neq j} G_{i,j}}$.

Podsumowując: jeśli promień spektralny macierzy A będzie mniejszy od jedności, wtedy wektor własny P_λ równania $AP_\lambda = \lambda P_\lambda$ spełni nierówność $AP_\lambda \leq P_\lambda$ jako wektor optymalnych poziomów głośności dla każdego stolika. Dodatkowo, rozwiązanie to jest właściwe tak długo, jak γ_i nie przewyższy poziomu $\frac{1}{\sum_{i \neq j} G_{i,j}}$. Jeżeli poziom ten zostanie przekroczyony, wówczas nie możemy być pewni, czy nasze rozwiązanie się sprawdzi. Warto wtedy pomyśleć o zmianie konfiguracji stolików.

3.3. Modelowanie populacji – model Lesliego

Modele populacji roślin, zwierząt czy też ludzi są typowymi przykładami nieujemnych systemów dynamicznych, w których zmienne położenia reprezentują biomasę, gęstość lub liczebność populacji. Wiele modeli, szczególnie te opisujące drapieżnictwo, konkurencję i symbiozę pomiędzy gatunkami, ma postać nielinijową – w takich przypadkach potrzebne są specyficzne narzędzia. Wartym uwagi wyjątkiem jest model Lesliego opisujący ewolucję populacji w czasie, w którym wskaźniki płodności i przeżycia poszczególnych jednostek są ściśle zależne od ich wieku.

W modelu Lesliego czas jest dyskretny, wyznaczając sezon reprodukcji (najczęściej rok w przypadku ssaków). Zmienne $x_1(t), x_2(t), \dots, x_n(t)$ będą tu opisywać liczbę samic (mogą to być też jednostki lub pary) w wieku $1, 2, \dots, n$ na początku roku t .

W najprostszym możliwym przypadku proces starzenia można opisać poniższym równaniem:

$$x_{i+1}(t+1) = s_i x_i(t), \quad i = 1, 2, \dots, n-1,$$

gdzie $s_i > 0$ to współczynnik przeżycia w wieku i (ta część samic w wieku i , która przeżyje przynajmniej rok). Pierwsze równanie stanu bierze pod uwagę proces reprodukcji:

$$x_1(t+1) = s_0(f_1 x_1(t) + f_2 x_2(t) + \dots + f_n x_n(t)),$$

gdzie $s_0 > 0$ to współczynnik przeżycia podczas pierwszego roku życia, $f_i \geq 0$ to wskaźnik płodności samic w wieku i (średnia liczba samic urodzona z każdej samicy w wieku i). Te równania, zaproponowane przez Lesliego, prowadzą do nieujemnego liniowego modelu autonomicznego:

$$x(t+1) = Ax(t),$$

gdzie macierz A (nazywana macierzą Lesliego) przedstawia się następująco:

$$\begin{bmatrix} s_0 f_1 & s_0 f_2 & \dots & s_0 f_{n-1} & s_0 f_n \\ s_1 & 0 & \dots & 0 & 0 \\ 0 & s_2 & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & s_{n-1} & 0 \end{bmatrix}.$$

Mimo że model Lesliego wydaje się być bardzo prosty, jest często używany do przeprowadzania prognozowania demograficznego:

$$x(k) = A^k x(0)$$

przy znanym stanie początkowym $x(0)$.

Skomentujmy jeszcze użyteczność powyższego modelu. W modelach Lesliego wskaźniki płodności i przeżycia zależą tylko od wieku. W rzeczywistości jest to prawdą pod warunkiem, że każda klasa wiekowa nie jest zbyt liczna. Faktycznie, jeśli tylko liczebność danej klasy rośnie, pojawiają się sytuacje, w których wskaźniki płodności lub przeżycia ulegają redukcji (na przykład, utrudnione jest znalezienie pożywienia lub przestrzeni do reprodukcji, z drugiej strony łatwiej o rozprzestrzenianie się epidemii, itd.). To oznacza, że modele Lesliego dobrze nadają się do opisu dynamiki populacji skazanych na wymarcie – tj. charakteryzujących się małymi wartościami $x_i(t)$ – możemy dla nich przyjąć, że ich wskaźniki płodności/przeżycia są stałe w czasie. Modele Lesliego są również bardzo skuteczne w krótkoterminowym prognozowaniu rosnących populacji. Wzbogacenie wiedzy na temat dynamiki populacji zwierząt jest ważne z uwagi na sformułowanie odpowiednich regulacji, np. dotyczących polowań.

Badając własności macierzy Lesliego możemy zauważać, że jest ona nieujemna i jeśli $f_n > 0$, to jest również niereductowalna. Można więc zastosować do niej twierdzenie Perrona–Frobeniusa. Znormalizowany wektor własny dla macierzy Lesliego (stwarzyszony z λ) jest nazywany stabilną strukturą wiekową – jest to

z grubsza asymptotyczna dystrybucja wieku w czasie. Jego współrzędne odpowiadają udziałom poszczególnych klas wiekowych w całej populacji. Natomiast λ możemy interpretować jako asymptotyczną stopę wzrostu populacji – stopę wzrostu w stanie stabilnej struktury wiekowej. Da nam to informację o tym, czy dana populacja będzie rosła bez ograniczenia ($\lambda > 1$), ulegnie wymarciu ($\lambda < 1$) czy będzie dążyć do stanu równowagi ($\lambda = 1$).

Rozważmy macierz Lesliego zdefiniowaną powyżej z $f_n > 0$ i oznaczmy wektor własny odpowiadający λ jako $x_\lambda \geq 0$. Wtedy zachodzi

$$A^k - \lambda^k P_1 \rightarrow 0,$$

gdy $k \rightarrow \infty$, gdzie P_1 jest projekcją na jednowymiarową podprzestrzeń rozpiętą przez x_λ .

Zauważmy, że w wielu modelach lepiej jest dzielić populację nie na równe klasy wiekowe, ale na tzw. grupy stanów (ang. *stage groups*). Taki typ modelowania jest często używany w przypadku gatunków długowiecznych, ponieważ dane dla konkretnych grup wiekowych nie są dostępne, a jednostkowe klasy wiekowe dla gatunków żyjących (jak czarne niedźwiedzie) np. do 30 lat, skutkowałyby macierzą 30×30 . Jako przykład⁴ rozważmy dziką populację czarnych niedźwiedzi w stanie Virginia w latach 1994–1999. Analiza statystyczna prowadzi do następujących danych zebranych w poniższej tabeli.

Klasa wiekowa [lata]	średni wskaźnik reprodukcji	średni wskaźnik przeżycia
0–1	0	0,8
1–2	0	0,75
2–3	0	0,71
3–4	0,28	0,84
4 i widż"cej	0,58	0,84

Macierz Lesliego dla powyższych danych (okazała się skuteczna w analizie przeprowadzonej przez biologów) przedstawia się następująco (przyjmujemy, że $s_0 = 1$):

$$\begin{bmatrix} 0 & 0 & 0 & 0,28 & 0,58 \\ 0,80 & 0 & 0 & 0 & 0 \\ 0 & 0,75 & 0 & 0 & 0 \\ 0 & 0 & 0,71 & 0 & 0 \\ 0 & 0 & 0 & 0,84 & 0,84 \end{bmatrix}.$$

Zwróćmy uwagę, że powyższa macierz różni się od klasycznej postaci macierzy Lesliego – wyraz $a_{5,5}$ jest niezerowy i przyjmuje wartość wskaźnika s_n przeżywalności ostatniej (dorosłej) grupy. Wynika to właśnie z podziału populacji na grupy stanów.

Dla rozważanego przypadku λ wynosi około 1,04. Znormalizowany wektor własny stwarzyszony z λ ma postać $x \approx (0,23; 0,18; 0,13; 0,09; 0,37)$. Populacja będzie zatem rosnąć i dążyć do rozkładu zadanego przez poszczególne wyrazy wektora x .

⁴Za: [7].

3.4. Page Rank

Algorytm Page Rank, opatentowany przez dwóch doktorantów Uniwersytetu Stanforda w 1998, Larry'ego Page'a oraz Sergey'a Brina, ma na celu określenie „wartości” stron internetowych i sporządzenie ich rankingu. Podstawowa wersja algorytmu sprowadza się do skorzystania z twierdzenia Perrona–Frobeniusa w analogiczny sposób, jak w przypadku naszego rankingu dowcipów. Tym razem w macierzy A jako „wygraną” będziemy określać istnienie odnośnika z jednej strony do drugiej.

DEFINICJA 30 (MACIERZ GOOGLE'A)

Macierzą Google'a nazywamy macierz $G = dS + (1 - d)E$, gdzie: $0 < d < 1$ to współczynnik tłumienia/znudzenia (ang. *damping factor*), S to lewa macierz Markowa, otrzymana z macierzy sąsiedztwa danego grafu przez przeskalowanie, $E = (e_{ij})$, gdzie $e_{ij} = \frac{1}{n}$ dla każdego i, j .

UWAGA 31 (O WSPÓŁCZYNNIKU TŁUMIENIA/ZNUDZENIA)

Według Brina i Page'a, „o algorytmie PageRank można myśleć jak o modelu zachowań użytkownika Internetu. Zakładamy, że istnieje losowy użytkownik, który kliką na linki, nigdy się nie cofając. Jednak w pewnym momencie nudzi się tym zachowaniem i zaczyna od innej, losowej strony. Prawdopodobieństwo, że ten przypadkowy użytkownik odwiedzi jakąś stronę to jej PageRank. Współczynnik tłumienia, to prawdopodobieństwo, że ten użytkownik znudzi się i zażąda innej, losowej strony. Jedna z ważnych zmian to dodanie współczynnika d do pojedynczej strony bądź do grupy stron. To pozwala na personalizację obliczeń i czyni wreszcie niemożliwym, by system zostać celowo zmyłyony, w celu uzyskania wyższej pozycji w rankingu. Oczywiście, istnieje wiele innych rozszerzeń algorytmu PageRank”.⁵

UWAGA 32 (O WSPÓŁCZYNNIKU TŁUMIENIA/ZNUDZENIA)

Według Brina i Page'a, „do obliczeń przyjmuje się zazwyczaj współczynnik d równy 0,85”.

UWAGA 33

Page i Brin potraktowali model zachować użytkownika Internetu jako proces stochastyczny, na podstawie którego zdefiniowali macierz Google'a będącą macierzą Markowa. W związku z tym stacjonarny wektor własny stwarzyszony z maksymalną wartością własną, na podstawie twierdzenia Perrona–Frobeniusa, jest jedyny. Możemy zatem utożsamiać go z pozycję rankingową.

DEFINICJA 34 (RÓWNANIE PAGERANK)

Równaniem PageRank nazywamy równanie:

$$Gv = v,$$

gdzie v to wektor własny macierzy G .

W rozwiniętej formie równanie PageRank przedstawia się następująco:

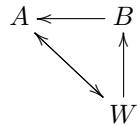
$$[dS + (1 - d)E]v = v.$$

⁵Za: [10].

PRZYKŁAD 35

Rozważmy 3 strony internetowe – A , B i W , gdzie A jest połączone z B i W (do strony A możemy dojść ze strony B lub ze strony W), B z W , a W z A . Znajdźmy ranking PageRank, jeśli $d = 0,9$.

Graf dla tej sytuacji został przedstawiony poniżej.



Latwo zauważyć”, iż macierz sąsiedztwa tego grafu to:

$$A = \begin{bmatrix} 0 & 1 & 1 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix},$$

a z kolei otrzymana z niej macierz stochastyczna wygląda następująco:

$$S = \begin{bmatrix} 0 & 1 & \frac{1}{2} \\ 0 & 0 & \frac{1}{2} \\ 1 & 0 & 0 \end{bmatrix}.$$

Wówczas macierz Google'a przedstawia się jak poniżej:

$$G = d \begin{bmatrix} 0 & 1 & \frac{1}{2} \\ 0 & 0 & \frac{1}{2} \\ 1 & 0 & 0 \end{bmatrix} + \begin{bmatrix} \frac{1-d}{3} & \frac{1-d}{3} & \frac{1-d}{3} \\ \frac{1-d}{3} & \frac{1-d}{3} & \frac{1-d}{3} \\ \frac{1-d}{3} & \frac{1-d}{3} & \frac{1-d}{3} \end{bmatrix}.$$

Podstawiając d , uzyskujemy

$$G = \begin{bmatrix} \frac{1}{30} & \frac{14}{15} & \frac{29}{60} \\ \frac{1}{30} & \frac{1}{30} & \frac{29}{60} \\ \frac{14}{15} & \frac{1}{30} & \frac{1}{30} \end{bmatrix}.$$

I obliczamy, zgodnie z równaniem PageRank, wektor własny v macierzy G związany z największą wartością własną tej macierzy: $\lambda = 1$. Wektor ten wynosi $v \approx (0,4; 0,21; 0,39)$, co znaczy, że największą wartość PageRank ma strona A . Wynik można ten interpretować w następujący sposób: przez około 40% czasu przypadkowy użytkownik odwiedza stronę A albo: z prawdopodobieństwem prawie 40% przypadkowy użytkownik trafi na stronę A .

3.5. Inne zastosowania

Poniżej wymienimy i pokróćce opiszymy inne zastosowania twierdzenia Perrona–Frobeniusa.

- **Epidemiologia:**

Wartość własna z twierdzenia Perrona–Frobeniusa determinuje tzw. próg Kermacka–McKendricka w pewnych modelach epidemiologicznych.

- **Modelowanie ekonomiczne:**
 - **Model Input–Output Leontiefa:**
ukazuje zależności pomiędzy różnymi gałeziami gospodarki. Szukamy takiego minimalnego wektora zaopatrzenia, który zaspokoi dany popyt na rynku.
 - **Prawo Walrasa o równowadze rynków konkurencyjnych:**
równowaga cenowa jest tu zdeterminowana przez wartość własną macierzy danego problemu
- **Topologia niskich wymiarów:** twierdzenie Perrona–Frobeniusa jest ważne dla klasyfikacji Thurstona homeomorfizmów powierzchni.
- **Iteracyjna analiza macierzy:**
Twierdzenie Steina-Rosenberga wykorzystuje twierdzenie Perrona–Frobeniusa do porównania wskaźników zbieżności dwóch metod iteracyjnych używanych do rozwiązywania równań liniowych – są to metody Gaussa–Seidela oraz Jacobięgo.

Literatura

- [1] <http://www.math.harvard.edu/library/sternberg/slides/1180912pf.pdf>
- [2] <https://mohitagrwal.files.wordpress.com/2010/02/presentation.pdf>
- [3] <http://facultypages.morris.umn.edu/math/Ma4901/Sp2014/Final/Final-AndrewLundborg.pdf>
- [4] http://www.math.upenn.edu/kazdan/312F12/JJ/MarkovChains/markov_google.pdf
- [5] <http://stat.wharton.upenn.edu/steele/Courses/956/Ranking/RankingFootballSIAM93.pdf>
- [6] <http://www.math.utah.edu/keener/lectures/rankings.pdf>
- [7] 17th Internet Seminar on Evolution Equations 2013/14: Positive Operator Semigroups and Applications, Andras Batkai, Marjeta Kramar Fijavz, Abdelaziz Rhandi, November 20, 2013
- [8] <http://infolab.stanford.edu/backrub/google.html>
- [9] <http://pubs.siam.org/doi/pdf/10.1137/S0036144599359449>
- [10] http://www.math.harvard.edu/knill/teaching/math19b_2011/handouts/lecture34.pdf

¹*Instytut Matematyki
Uniwersytet Jagielloński w Krakowie
ul. Gołębia 24
31-007 Kraków
E-mail: barbara.ciesielska@student.uj.edu.pl*

²*Instytut Matematyki
Uniwersytet Jagielloński w Krakowie
ul. Gołębia 24
31-007 Kraków
E-mail: agnes.kowalczyk@student.uj.edu.pl*

Przysłano: 31.05.2015; publikacja on-line: 31.08.2015.

TEORETYCZNE PODSTAWY INFORMATYKI

21/11/2016

WFAiS UJ, Informatyka Stosowana
I rok studiów, I stopień

Wykład 9

2

Dane w
postaci
grafów

- ❑ Dane w postaci grafów
- ❑ Algorytm PageRanking

Przykład: social network

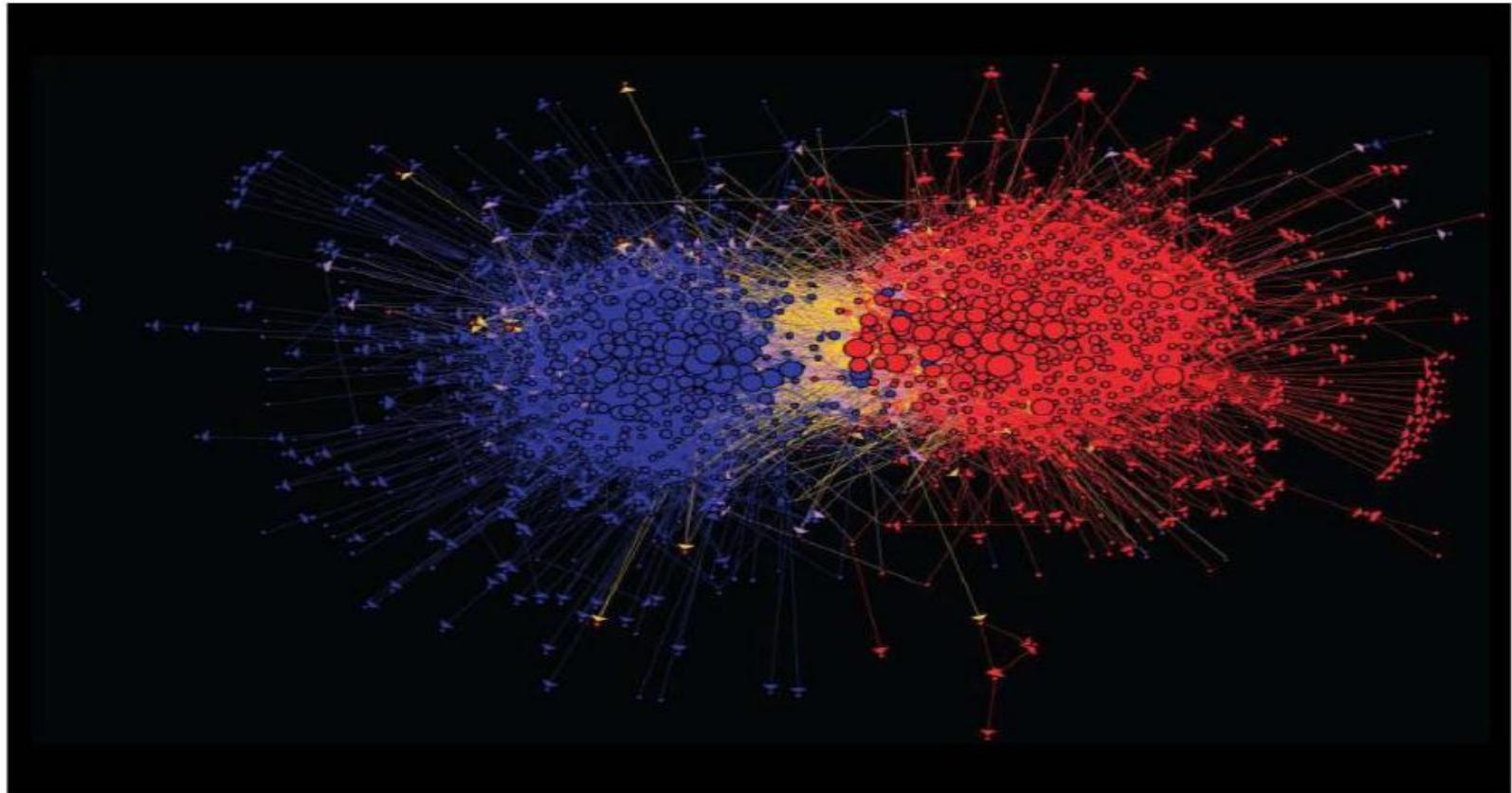
3



Facebook social graph

Przykład: media network

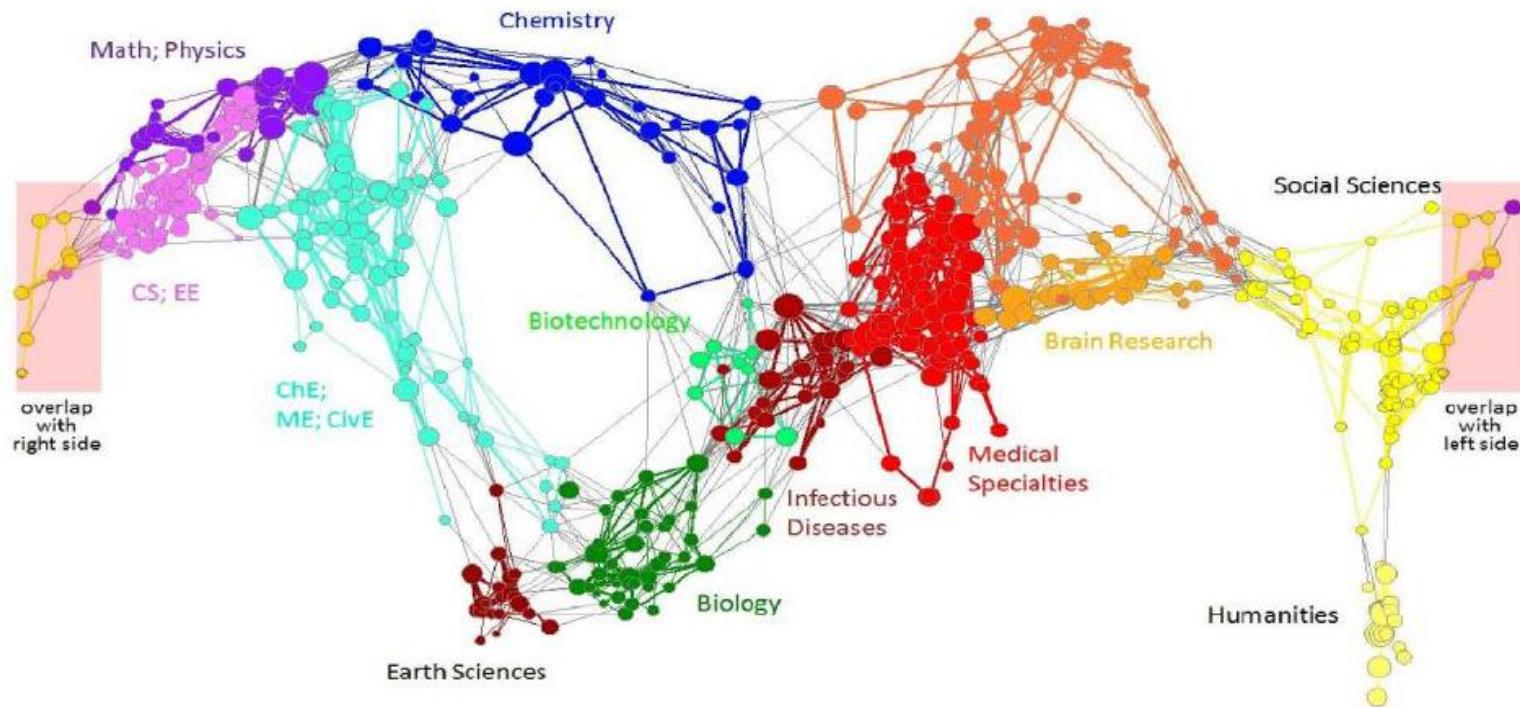
4



Connections between political blogs
Polarization of the network [Adamic-Glance, 2005]

Przykład: information network

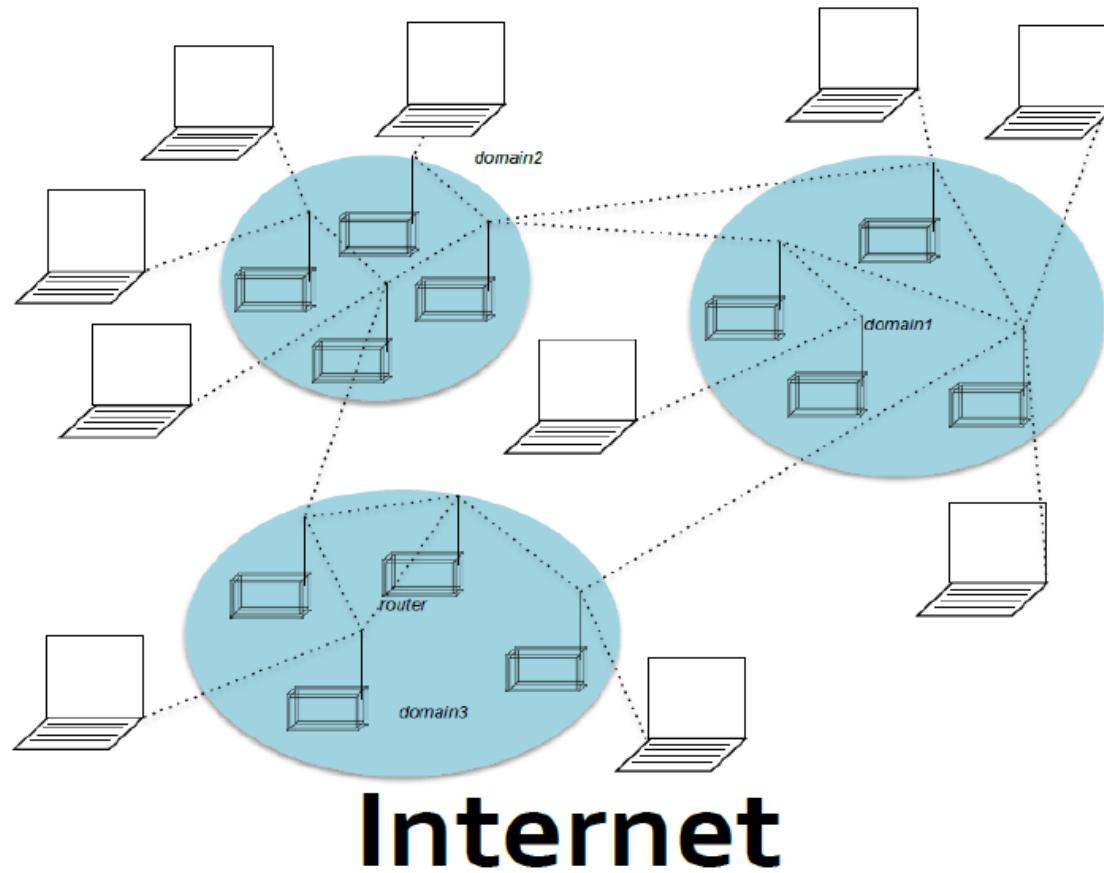
5



Citation networks and Maps of science
[Börner et al., 2012]

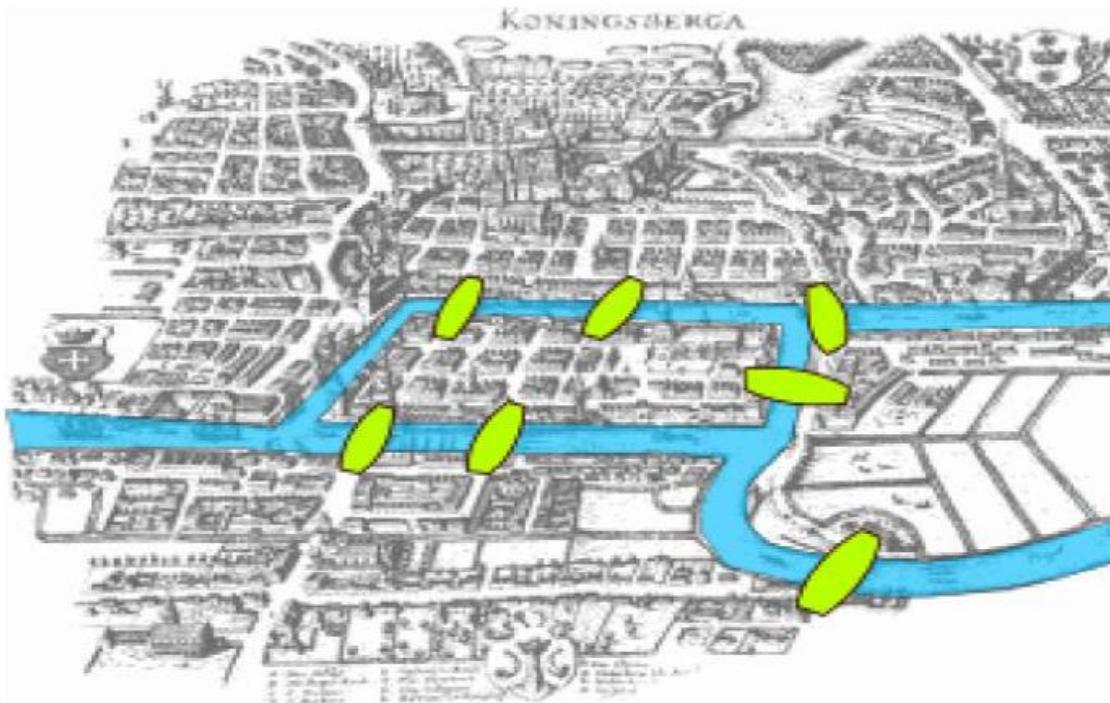
Przykład: communication network

6



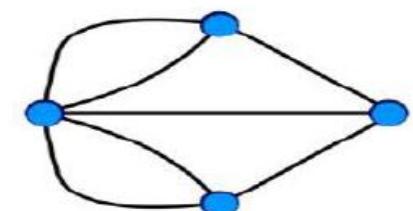
Przykład: technological network

7



Seven Bridges of Königsberg
[Euler, 1735]

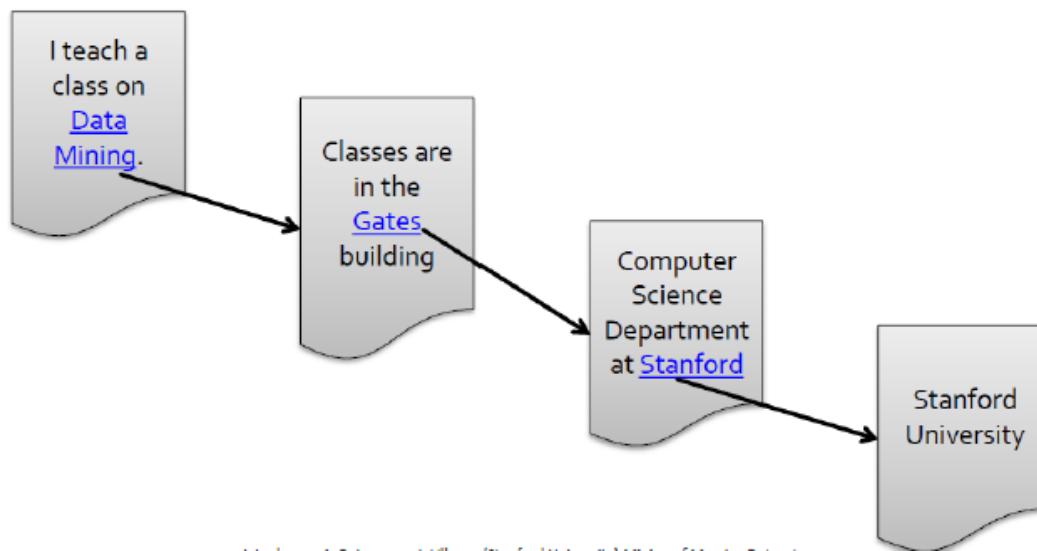
Return to the starting point by traveling each link of the graph once and only once.



Web jako graf

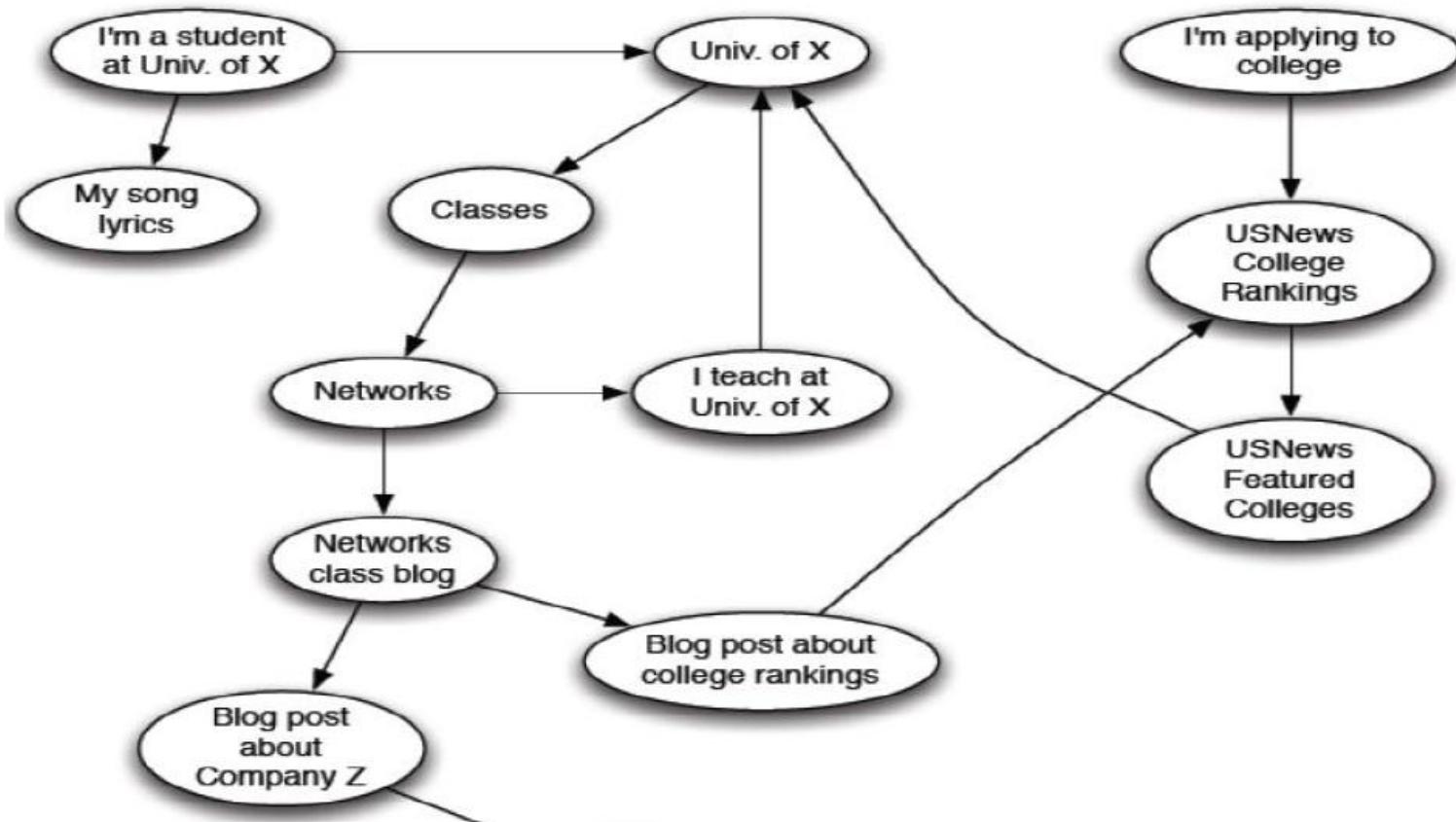
8

- Web jako skierowany graf
 - Węzły: strony internetowe
 - Strzałki: hyperlinki



Web jako skierowany graf

9



Jak organizować web

10

- Pierwsza próba: ręcznie tworzone katalogi
 - ▣ Yahoo, DMOZ, LookSmart
- Następna próba: WebSearch
 - ▣ Przeszukiwanie zawartości stron z małych i wiarygodnych podzbiorów: artykuły w gazetach, patenty, etc.
 - ▣ Ale: Web jest ogromny, wiarygodność stron trudna do weryfikacji, spamowanie web, itd.

Podstawowe wyzwania

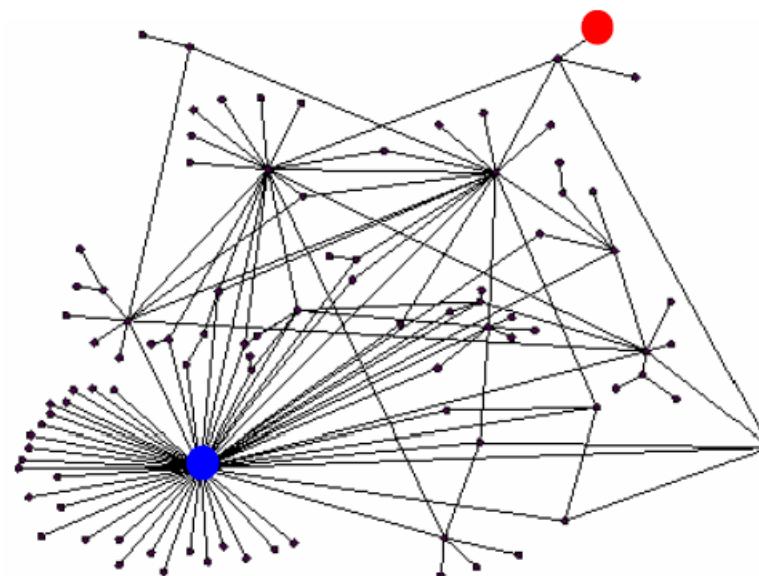
11

- W jaki sposób sprawdzać że strona jest wiarygodna?
- Jaka jest najlepsza odpowiedź na hasło „gazeta”?
 - ▣ Nie ma jednej najlepszej odpowiedzi
 - ▣ Strony które pokazują informacje na temat gazet mogą dotyczyć wielu różnych gazet

Które strony są najważniejsze

12

- Nie wszystkie strony są tak samo ważne
- Zróbmy „ranking” stron ze względu na ilość linków



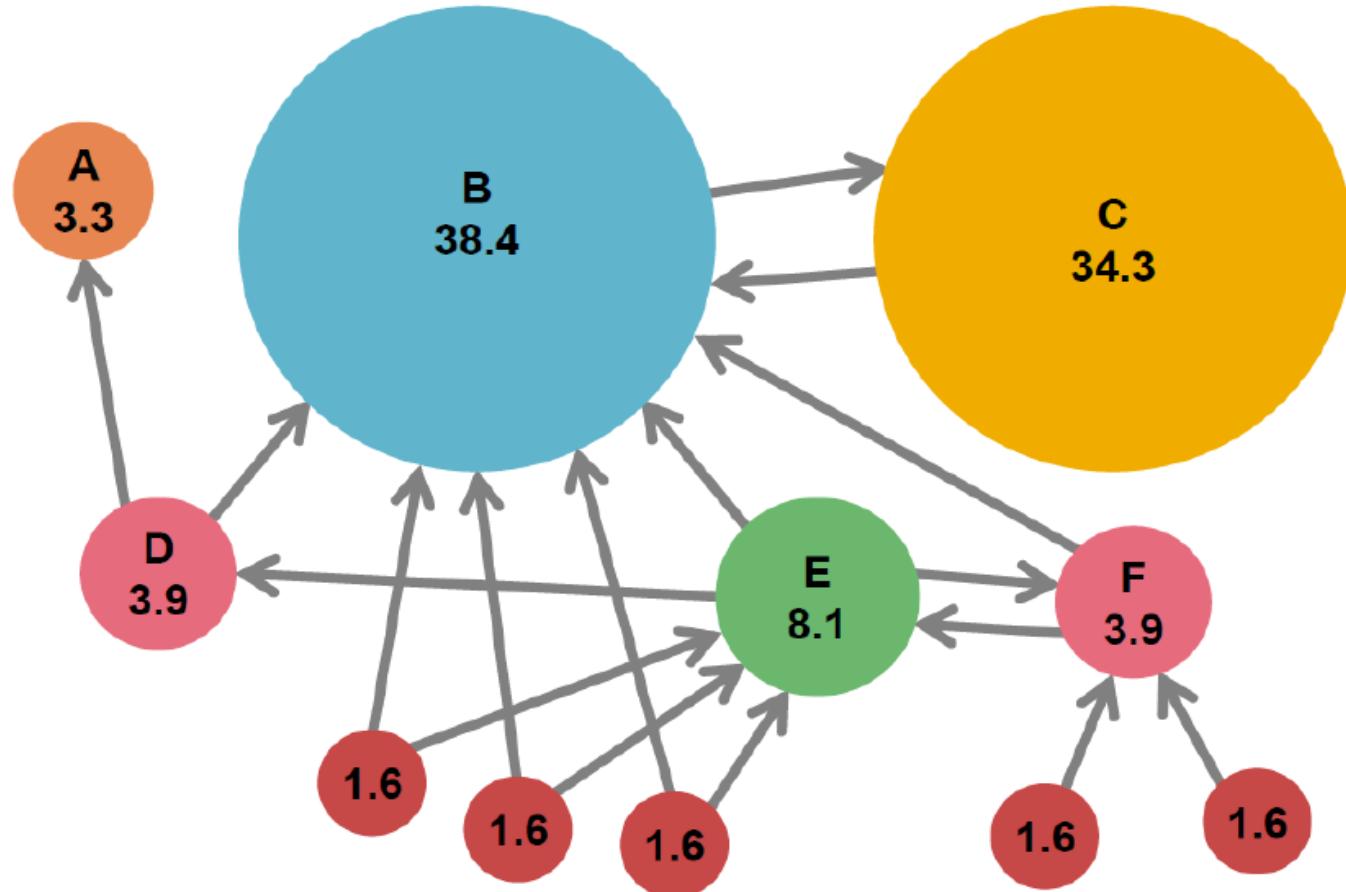
Algorytmy do analizy linków

13

- Wiele różnych algorytmów zostało rozwiniętych
- Omówię algorytm PageRank używany przez Google.
 - ▣ Każdy link liczy się jako punkt, bardziej ważna strona to ta która ma więcej linków:
Wchodzących? Wychodzących?
 - ▣ A linki a ważnych stron do danej strony powinny się liczyć bardziej! Hm.. To wygląda jak pytanie rekurencyjne..

Przykład: PageRank punktacja

14

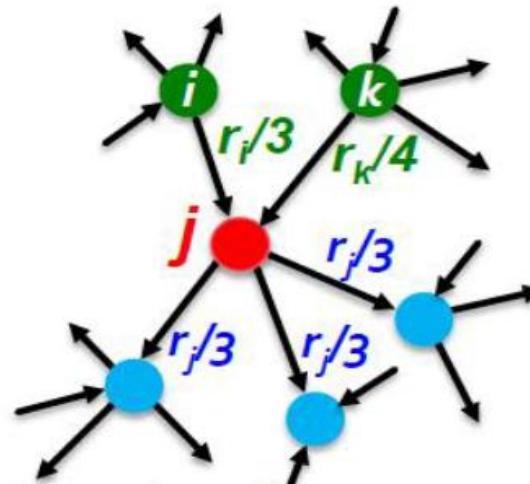


Prosta rekurencyjna formuła

15

- Wartość każdego linku jest proporcjonalna do wartości strony z której wychodzi
- Jeżeli strona **j** z wartością r_j na **n**-wychodzących linków to każdy ma wartość r_j/n .
- Wartość strony **j** jest sumą wartości linków do niej wchodzących.

$$r_j = r_i/3 + r_k/4$$



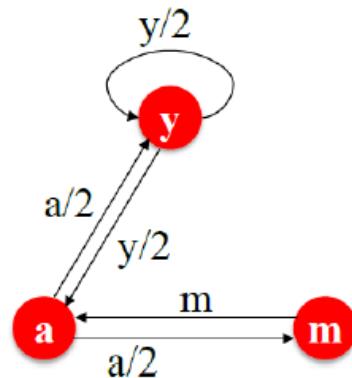
Page Rank: flow model

16

- Link z „ważnej” strony jest więcej warty
- Strona jest „ważna” jeżeli jest wskazywana przez wiele innych stron.
- Zdefiniujemy „rank” strony

$$r_j = \sum_{i \rightarrow j} \frac{r_i}{d_i}$$

d_i ... out-degree of node i



“Flow” equations:

$$r_y = r_y/2 + r_a/2$$

$$r_a = r_y/2 + r_m$$

$$r_m = r_a/2$$

Rozwiązywanie dla „flow” równania

17

- 3 równania, 3 niewiadome, nie ma stałej

- Nie ma jednego rozwiązania
- Dodatkowy warunek dot. Normalizacji

Flow equations:

$$r_y = r_y/2 + r_a/2$$

$$r_a = r_y/2 + r_m$$

$$r_m = r_a/2$$

- $r_y + r_a + r_m = 1$

- **Solution:** $r_y = \frac{2}{5}$, $r_a = \frac{2}{5}$, $r_m = \frac{1}{5}$

- Układ równań możemy rozwiązać np. metodą eliminacji Gaussa. Ale potrzebujemy czegoś lepszego

Interpretacja macierzowa

18

- Ponieważ kolumny normalizowane do 1 => interpretacja prawdopodobieństwowa.
 - ▣ Page i ma d_i out-link
 - ▣ Jeżeli $i \rightarrow j$, to $M_{ji} = 1/d_i$, w pozostałych $M_{ji} = 0$
 - ▣ Kolumny tej macierzy sumują się do 1
- Oznaczmy $r_i = \text{rank strony}$ $\sum_i r_i = 1$
- Równanie „flow”

$$\mathbf{r} = \mathbf{M} \cdot \mathbf{r}$$

$$r_j = \sum_{i \rightarrow j} \frac{r_i}{d_i}$$

Przykład

19

$$\begin{matrix} & i \\ j & \downarrow \\ \text{---} & \downarrow \\ 1/3 & \downarrow \end{matrix} \quad \cdot \quad \begin{matrix} r_i \\ = \\ r_j \end{matrix}$$

Diagram illustrating matrix multiplication. A 3x3 matrix M is shown with its columns labeled i , j , and $1/3$. The matrix has three columns of equal width. The first column i is highlighted in green. The second column j is highlighted in purple. The third column is labeled $1/3$. To the right of the matrix, the expression $\cdot r$ is shown, followed by an equals sign and a green bar divided into two segments, labeled r_i and r_j .

Power iteration

20

- Jak efektywnie możemy rozwiązać takie równanie?

NOTE: x is an eigenvector with the corresponding eigenvalue λ if:

$$Ax = \lambda x$$

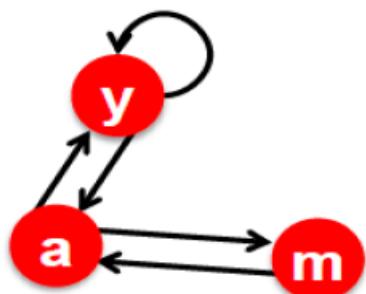
- Metoda „power iteration”

$$\mathbf{r} = \mathbf{M} \cdot \mathbf{r}$$

- Wektor \mathbf{r} jest wektorem własnym macierzy prawdopodobieństwa.

Przykład

21



	y	a	m
y	$\frac{1}{2}$	$\frac{1}{2}$	0
a	$\frac{1}{2}$	0	1
m	0	$\frac{1}{2}$	0

$$\mathbf{r} = \mathbf{M} \cdot \mathbf{r}$$

$$\mathbf{r}_y = \mathbf{r}_y/2 + \mathbf{r}_a/2$$

$$\mathbf{r}_a = \mathbf{r}_y/2 + \mathbf{r}_m$$

$$\mathbf{r}_m = \mathbf{r}_a/2$$

$$\begin{bmatrix} y \\ a \\ m \end{bmatrix} = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} & 0 \\ \frac{1}{2} & 0 & 1 \\ 0 & \frac{1}{2} & 0 \end{bmatrix} \begin{bmatrix} y \\ a \\ m \end{bmatrix}$$

Metoda „power iteration”

22

- Mając dany graf z N węzłami, gdzie każdy węzeł to są strony a skierowane krawędzie to są hiperlinki
 - ▣ Zainicjalizuj $r(0) = [1/N, \dots, 1/N]^T$
 - ▣ Iteruj: $r^{(t+1)} = M r^{(t)}$
 - ▣ Zatrzymaj jeżeli $|r^{(t+1)} - r^{(t)}|_1 < \varepsilon$

$|\mathbf{x}|_1 = \sum_{1 \leq i \leq N} |x_i|$ is the **L₁** norm

$$r_j^{(t+1)} = \sum_{i \rightarrow j} \frac{r_i^{(t)}}{d_i}$$

d_i out-degree of node i

Przykład

23

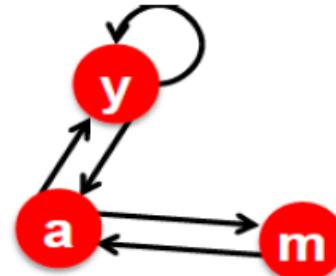
■ Power Iteration:

- Set $r_j = 1/N$
- 1: $r'_j = \sum_{i \rightarrow j} \frac{r_i}{d_i}$
- 2: $r = r'$
- If not converged: goto 1

■ Example:

$$\begin{bmatrix} r_y \\ r_a \\ r_m \end{bmatrix} = \begin{array}{cccccc} 1/3 & 1/3 & 5/12 & 9/24 & 6/15 \\ 1/3 & 3/6 & 1/3 & 11/24 & \dots & 6/15 \\ 1/3 & 1/6 & 3/12 & 1/6 & & 3/15 \end{array}$$

Iteration 0, 1, 2, ...



	y	a	m
y	1/2	1/2	0
a	1/2	0	1
m	0	1/2	0

$$\mathbf{r}_y = \mathbf{r}_y/2 + \mathbf{r}_a/2$$

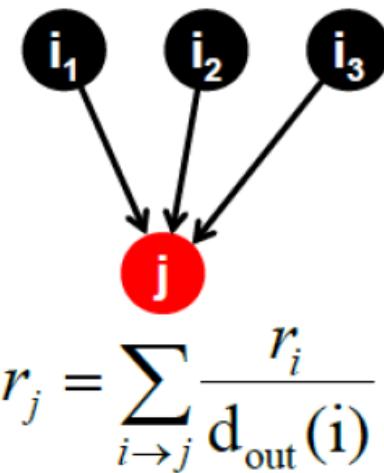
$$\mathbf{r}_a = \mathbf{r}_y/2 + \mathbf{r}_m$$

$$\mathbf{r}_m = \mathbf{r}_a/2$$

Błędzenie przypadkowe

24

- Oglądamy przypadkowe strony internetowe
 - ▣ W momencie **t**, jesteśmy na stronie **i**
 - ▣ W czasie **(t+1)** przechodzimy losowo na jedną ze stron podłączonych ze stroną **i**
 - ▣ W pewnym momencie kończymy na stronie **j**
 - ▣ Powtarzamy ten krok nieskończoną ilość razy.



Rozkład stacjonarny

25

- Na jakiej stronie jesteśmy w czasie $(t+1)$?

- ▣ Błędzimy losowo

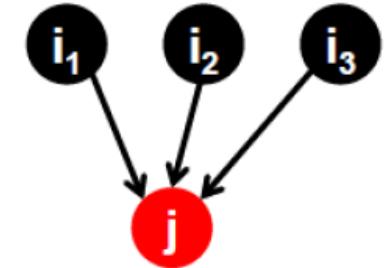
$$p(t+1) = M \cdot p(t)$$

- ▣ Założmy że rozwiążanie jest stacjonarne

$$p(t+1) = M \cdot p(t) = p(t)$$

- ▣ Nasze orginalne równanie to było

$$r = M \cdot r$$



$$p(t+1) = M \cdot p(t)$$

Procesy Markowa

26

- Dla grafów które spełniają pewne warunki rozwiążanie stacjonarne będzie osiągnięte niezależnie od warunków początkowych.
- To rozwiązanie jest jednoznaczne.

PageRank wg. Googla

27

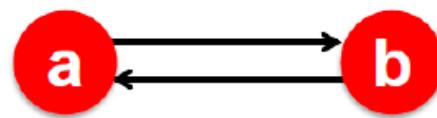
$$r_j^{(t+1)} = \sum_{i \rightarrow j} \frac{r_i^{(t)}}{d_i} \quad \text{or equivalently} \quad r = Mr$$

- Trzy pytania:
 - Czy rozwiązanie zbieżne?
 - Czy zbieżne do rozwiązania które oczekujemy?
 - Czy rozwiązanie jest rozsądne?

Czy rozwiążanie jest zbieżne?

28

- „Spider trap” problem



$$r_j^{(t+1)} = \sum_{i \rightarrow j} \frac{r_i^{(t)}}{d_i}$$

■ Example:

$$\begin{matrix} r_a \\ r_b \end{matrix} = \begin{matrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{matrix}$$

Iteration 0, 1, 2, ...

Czy zbieżne do tego czego oczekujemy?

29

- „Dead end” problem



■ Example:

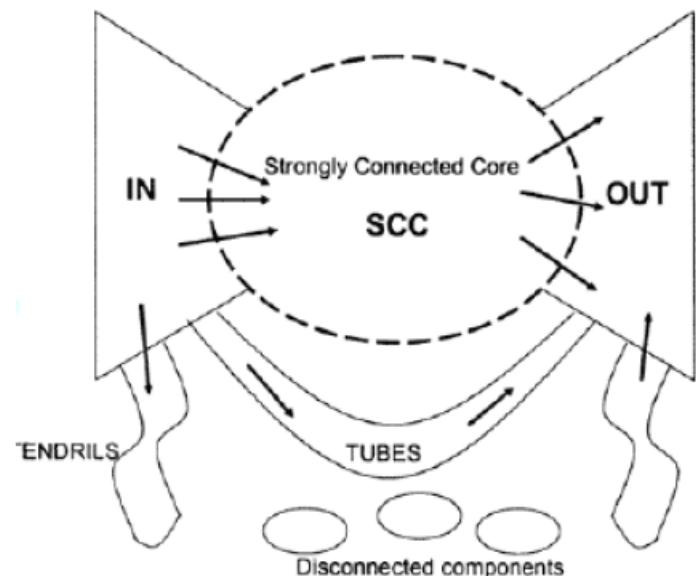
$$\begin{array}{lcl} r_a & = & 1 & 0 & 0 & 0 \\ r_b & & 0 & 1 & 0 & 0 \end{array}$$

Iteration 0, 1, 2, ...

PageRank: problemy

30

- Niektóre strony są „dead end” czyli nie mają wychodzących linków
 - ▣ Takie strony powodują „wyciekanie” informacji o ważności stron
- Niektóre grupy stron tworzą „spider traps” czyli wszystkie out-linki są do zamkniętej grupy stron.
 - ▣ I wtedy tak grupa stron zaasimiluje wszystkie punkty ważności stron.

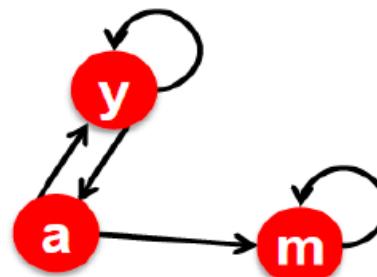


Spider Traps

31

■ Power Iteration:

- Set $r_j = 1$
- $r_j = \sum_{i \rightarrow j} \frac{r_i}{d_i}$
- And iterate



	y	a	m
y	$\frac{1}{2}$	$\frac{1}{2}$	0
a	$\frac{1}{2}$	0	0
m	0	$\frac{1}{2}$	1

$$\mathbf{r}_y = \mathbf{r}_y / 2 + \mathbf{r}_a / 2$$

$$\mathbf{r}_a = \mathbf{r}_y / 2$$

$$\mathbf{r}_m = \mathbf{r}_a / 2 + \mathbf{r}_m$$

■ Example:

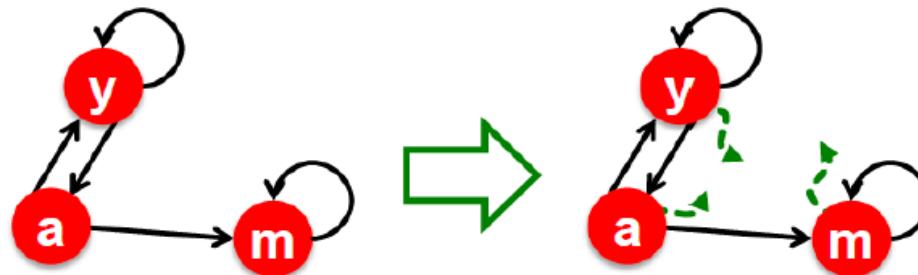
$$\begin{bmatrix} r_y \\ r_a \\ r_m \end{bmatrix} = \begin{matrix} 1/3 & 2/6 & 3/12 & 5/24 & \dots & 0 \\ 1/3 & 1/6 & 2/12 & 3/24 & \dots & 0 \\ 1/3 & 3/6 & 7/12 & 16/24 & \dots & 1 \end{matrix}$$

Iteration 0, 1, 2, ...

Rozwiązywanie: teleportacja

32

- W każdym kroku błądzenia można
 - ▣ z prawdopodobieństwem β pójść jedną z wychodzących ścieżek
 - ▣ z prawdopodobieństwem $(1 - \beta)$ przeskoczyć na losowo wybraną stronę
 - ▣ Najczęściej stosowane wartości $\beta = 0.8-0.9$

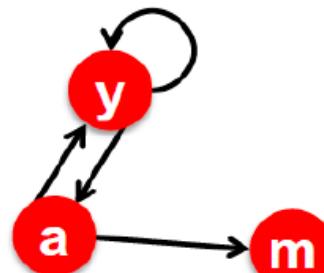


Problem: dead end

33

■ Power Iteration:

- Set $r_j = 1$
- $r_j = \sum_{i \rightarrow j} \frac{r_i}{d_i}$
- And iterate



	y	a	m
y	$\frac{1}{2}$	$\frac{1}{2}$	0
a	$\frac{1}{2}$	0	0
m	0	$\frac{1}{2}$	0

$$r_y = r_y/2 + r_a/2$$

$$r_a = r_y/2$$

$$r_m = r_a/2$$

■ Example:

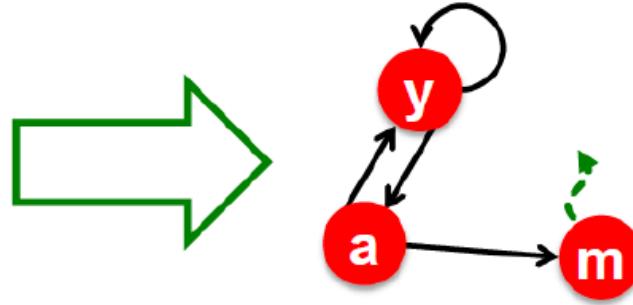
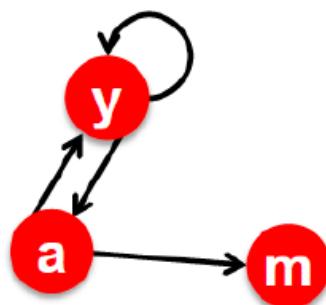
$$\begin{bmatrix} r_y \\ r_a \\ r_m \end{bmatrix} = \begin{matrix} 1/3 & 2/6 & 3/12 & 5/24 & \dots & 0 \\ 1/3 & 1/6 & 2/12 & 3/24 & \dots & 0 \\ 1/3 & 1/6 & 1/12 & 2/24 & \dots & 0 \end{matrix}$$

Iteration 0, 1, 2, ...

Rozwiążanie teleportacja

34

- Zawsze przeskocz do losowo wybranej strony
- Odpowiednio zmodyfikuj macierz przejść



	y	a	m
y	$\frac{1}{2}$	$\frac{1}{2}$	0
a	$\frac{1}{2}$	0	0
m	0	$\frac{1}{2}$	0

	y	a	m
y	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{3}$
a	$\frac{1}{2}$	0	$\frac{1}{3}$
m	0	$\frac{1}{2}$	$\frac{1}{3}$

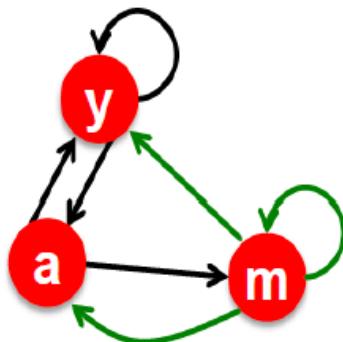
Teleportacja

35

- Powoduje że macierz staje się stochastyczna

$$A = M + a^T \left(\frac{1}{n} e \right)$$

- $a_{i...}=1$ if node i has out deg 0, =0 else
- e ...vector of all 1s



	y	a	m
y	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{3}$
a	$\frac{1}{2}$	0	$\frac{1}{3}$
m	0	$\frac{1}{2}$	$\frac{1}{3}$

$$r_y = r_y/2 + r_a/2 + r_m/3$$

$$r_a = r_y/2 + r_m/3$$

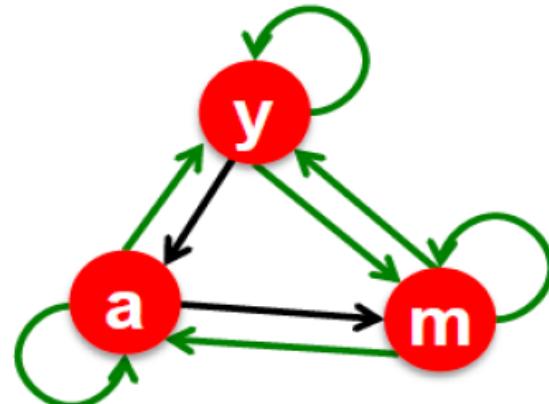
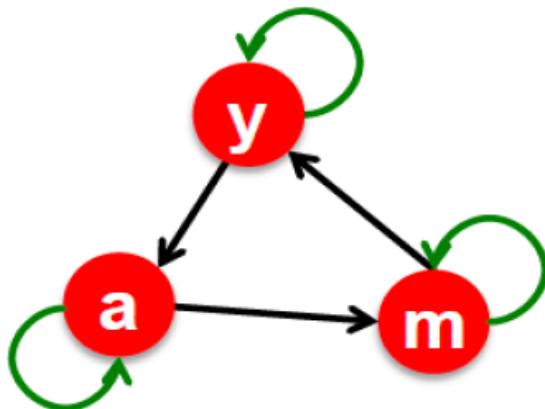
$$r_m = r_a/2 + r_m/3$$

Teleportacja

36

- Powoduje że macierz staje się aperiodyczna i niereduksowalna

Niezerowe prawdopodobieństwo
Przejścia z każdego stanu do
każdego innego.



Rozwiążanie Googla

37

PageRank equation [Brin-Page, 98]

$$r_j = \sum_{i \rightarrow j} \beta \frac{r_i}{d_i} + (1 - \beta) \frac{1}{n}$$

d_i ... out-degree
of node i

The above formulation assumes that M has no dead ends. We can either preprocess matrix M (**bad!**) or explicitly follow random teleport links with probability 1.0 from dead-ends.

Rozwiążanie Googla

38

- **PageRank equation** [Brin-Page, 98]

$$r_j = \sum_{i \rightarrow j} \beta \frac{r_i}{d_i} + (1 - \beta) \frac{1}{n}$$

- **The Google Matrix A :**

$$A = \beta M + (1 - \beta) \frac{1}{n} \mathbf{e} \cdot \mathbf{e}^T$$

\mathbf{e} ...vector of all 1s

- **A is stochastic, aperiodic and irreducible, so**

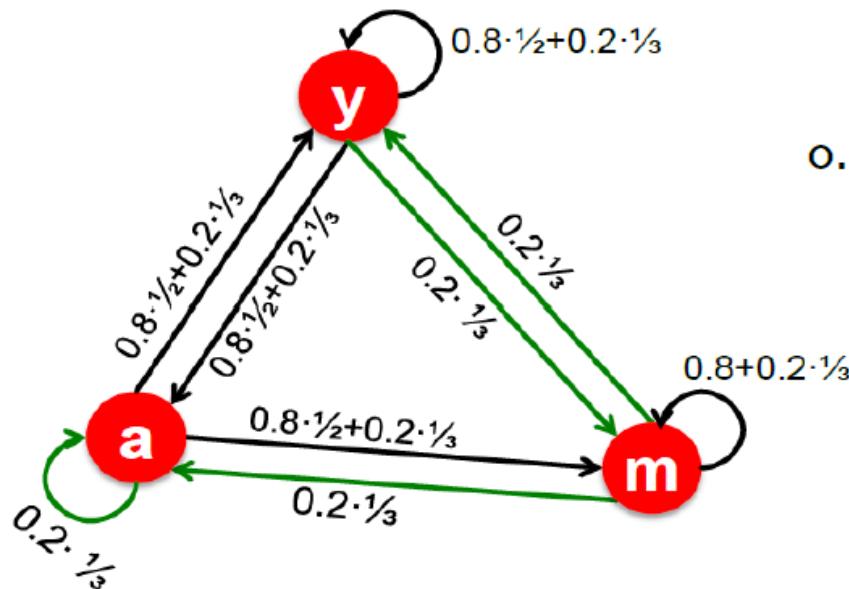
$$\mathbf{r}^{(t+1)} = A \cdot \mathbf{r}^{(t)}$$

- **What is β ?**

- In practice $\beta = 0.8, 0.9$ (make 5 steps and jump)

Przykład

39



$$\begin{array}{c}
 M \\
 \begin{array}{ccc} 1/2 & 1/2 & 0 \\ 1/2 & 0 & 0 \\ 0 & 1/2 & 1 \end{array}
 \end{array}
 \quad
 \begin{array}{c}
 0.8 \\
 + 0.2
 \end{array}
 \quad
 \begin{array}{c}
 1/n \cdot \mathbf{1} \cdot \mathbf{1}^T \\
 \begin{array}{ccc} 1/3 & 1/3 & 1/3 \\ 1/3 & 1/3 & 1/3 \\ 1/3 & 1/3 & 1/3 \end{array}
 \end{array}$$

$$\begin{array}{c}
 A \\
 \begin{array}{ccc} 7/15 & 7/15 & 1/15 \\ 7/15 & 1/15 & 1/15 \\ 1/15 & 7/15 & 13/15 \end{array}
 \end{array}$$

y	1/3	0.33	0.24	0.26	...	7/33
a	1/3	0.20	0.20	0.18	...	5/33
m	1/3	0.46	0.52	0.56		21/33

PageRank: kompletny algorytm

40

■ Output: PageRank vector r

- **Set:** $r_j^{(0)} = \frac{1}{N}, \quad t = 1$
- **do:**
 - $\forall j: r_j'^{(t)} = \sum_{i \rightarrow j} \beta \frac{r_i^{(t-1)}}{d_i}$
 - $r_j'^{(t)} = \mathbf{0}$ if in-deg. of j is 0
 - **Now re-insert the leaked PageRank:**
$$\forall j: r_j^{(t)} = r_j'^{(t)} + \frac{1-s}{N} \quad \text{where: } s = \sum_j r_j'^{(t)}$$
 - $t = t + 1$
- **while** $\sum_i |r_i^{(t)} - r_i^{(t-1)}| > \varepsilon$

INŻYNIERIA WIEDZY

KNOWLEDGE ENGINEERING (KE)

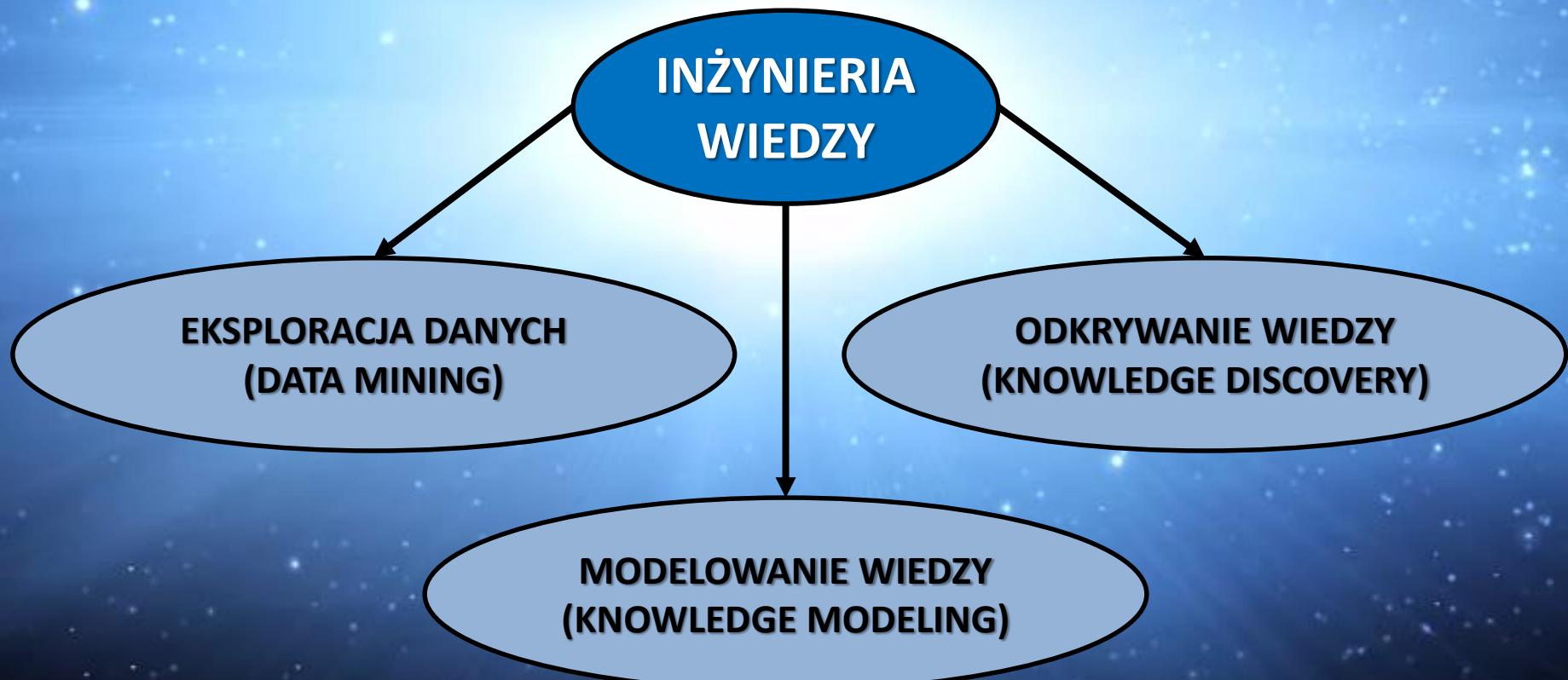


to obszar informatyki zajmujący się metodami eksploracji, reprezentacji i modelowania wiedzy z danych (ich zbiorów, reguł, baz danych) oraz metodami wnioskowania na ich podstawie.

EKSPLORACJA WIEDZY Z DANYCH

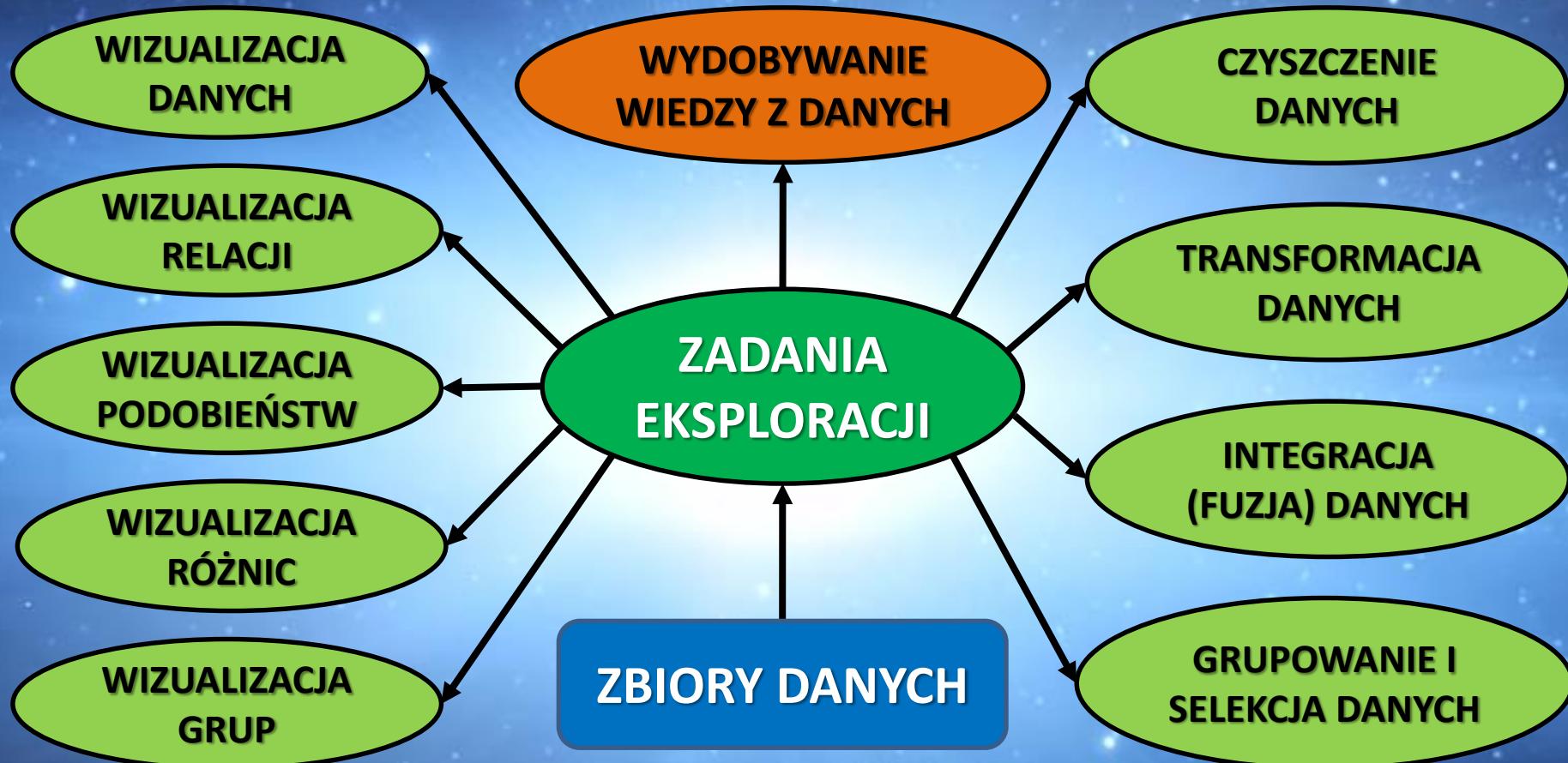
KNOWLEDGE DISCOVERY IN DATABASES (KDD)

to proces odkrywania wiedzy ukrytej w danych lub ich zbiorach (czyli baza danych) polegający na wyszukiwaniu prawidłowości, powtarzalności i zależności (relacji) pomiędzy danymi.



ZADANIA EKSPLORACJI DANYCH

DATA MINING TASKS



Eksploracja danych to zwykle proces wieloetapowy związany z wstępna obróbką danych (czyszczenie, normalizacja, standaryzacja lub inny rodzaj transformacji), porównywaniem, integracją, grupowaniem i selekcją danych oraz wizualizacją danych, ich cech, grup, podobieństw, różnic i zależności (relacji).

NARZĘDZIA EKSPLORACJI DANYCH

DATA MINING METHODS AND TOOLS



Eksploracja danych to zwykle proces wieloetapowy związany z wstępnią obróbką danych (czyszczenie, normalizacja, standaryzacja lub inny rodzaj transformacji), porównywaniem, integracją, grupowaniem i selekcją danych oraz wizualizacją danych, ich cech, grup, podobieństw, różnic i zależności (relacji).

KLASYFIKACJA - CLASSIFICATION

- to zadanie przyporządkowania wzorca do pewnej klasy.
- to zadanie rozpoznawania wzorca jako elementu pewnej klasy.

Klasa – to pewna grupa wzorców charakteryzujących się podobnymi cechami/właściwościami dla określających je atrybutów/parametrów.

W wyniku klasyfikacji wzorcowi zostaje przyporządkowana pewna klasa, reprezentowana zwykle przez pewną etykietę klasy.

Jeśli wzorzec należy równocześnie do kilku klas, wtedy mówimy o zagadnieniu multiklasyfikacji (multiclass classification), np.:

- ser Mozzarella należy do klas: serów, nabiału, produktów spożywczych

Sklasyfikowanie wzorca jako przynależnego do określonej klasy może być rozważane jako proces:

- Binarny / zero-jedynkowy / dyskretny: należy lub nie należy do klasy
- Rozmyty / predyktywny / ciągły: o określonym stopniu przynależności do klasy

REDUKCJA I TRANSFORMACJA DANYCH

Metody redukcji i transformacji danych – mają za zadanie doprowadzić do optymalnej reprezentacji dużych ilości danych, tj. takiej reprezentacji, żeby dane w dalszym ciągu były reprezentatywne dla rozważanego problemu, np. klasyfikacji, czyli umożliwiały poprawną dyskryminację wzorców, tj. rozróżnienie ich według pozostałych po redukcji danych.

Optymalna reprezentacja danych może być osiągnięta na skutek:

- **Redukcji wymiaru danych** – czyli usuwania mniej istotnych atrybutów danych, oraz **Selekcji** atrybutów najistotniejszych pod kątem rozwiązywanego zadania.
- **Transformacji danych** – czyli przekształcenia danych do innej, bardziej oszczędnej lub mniej wymiarowej postaci, która dalej pozwala na ich poprawne rozróżnianie i przetwarzanie, np.:
 - metoda analizy głównych składowych (PCA – Principal Component Analysis)
 - metoda analizy składowych niezależnych (ICA – Independent Component Analysis)
- **Agregacji i Asocjacji danych (Aggregate & Associate)** – czyli takiej reprezentacji danych, która polega na zagregowaniu reprezentacji takich samych i/lub podobnych danych i ich grup oraz ich odpowiednim do rozwiązywanego zadania powiązaniu w celu przyspieszenia ich przeszukiwania i przetwarzania.

WIZUALIZACJA I PREZENTACJA VISUALIZATION & PRESENTATION

to zadania związane z graficzną reprezentacją danych w takiej postaci, żeby zaprezentować dane w taki sposób, aby możliwe było:

- porównanie liczności danych określonego typu/grupy/zbioru/klasy,
- wskazanie zależności (relacji) pomiędzy danymi i ich grupami,
- wskazanie minimów, maksimów, średnich, odchyleń i wariancji danych,
- wskazanie rozkładów, agregacji, środków ciężkości,
- wskazanie podobieństw i różnic pomiędzy danymi i ich grupami,
- wskazanie reprezentantów, typowych i nietypowych danych,
- wskazanie wzorców lub wartości odstających od przeciętnych (outlier), błędnych, brakujących lub szczególnych,
- podział, odfiltrowanie lub selekcja pewnej grupy wzorców,
- oceny pokrycia przestrzeni danych i ich reprezentatywności dla zadania,
- oceny jakości, zaszumienia, poprawności, dokładności i pełności danych.

GŁÓWNE ETAPY EKSPLORACJI DANYCH

1. Zrozumienie zadania i **zdefiniowanie celu praktycznego eksploracji**, czyli przyporządkowanie zadania do grupy: klasyfikacji, grupowania, predykci lub asocjacji.
2. **Przygotowanie bazy danych** do analizy poprzez wyselekcjonowanie rekordów z baz danych najlepiej charakteryzujących rozważany problem.
3. **Czyszczenie i wstępna transformacja** danych poprzez ich normalizację, standaryzację, usuwanie danych odstających, usuwanie lub uzupełnianie niekompletnych wzorców.
4. **Transformacja danych** z postaci symbolicznej na postać numeryczną poprzez przypisanie im wartości lub rozmywanie (fuzzification) w zależności od stosowanej metody ich dalszego przetwarzania.
5. **Redukcja wymiaru danych i selekcja** najbardziej znaczących i dyskryminujących cech pozwalających uzyskać najlepsze zdolności uogólniające projektowanego systemu.
6. **Wybór techniki i metody** eksploracji danych na podstawie możliwości danej metody oraz rodzaju i liczności danych: numeryczne, symboliczne, sekwencyjne...
7. **Wybór algorytmu lub aplikacji** implementującej wybraną technikę eksploracji danych oraz **określenie optymalnych parametrów adaptacji/uczenia** wybranej metody (przydatne mogą tutaj być metody ewolucyjne, genetyczne, walidacja krzyżowa).
8. **Przeprowadzenie procesu konstrukcji, adaptacji lub uczenia** wybraną metodą.
9. **Eksplotacja systemu:** wnioskowanie, określanie grup, podobieństw, różnic, zależności, następstwa, implikacji.
10. **Douczanie systemu** na nowych danych lub utrwalanie zebranych wniosków z eksploracji.

PODSTAWOWE POJĘCIA I TERMINOLOGIA

Asocjacja – to proces stowarzyszenia ze sobą dwu lub więcej obserwacji.

W najprostszej postaci opisywana jest często przez reguły asocjacyjne.

Asocjacje są również postawą działania ludzkiego mózgu, pamięci i inteligencji, więc mogą być reprezentowane przez skomplikowane sieci neuronowe.

Atrybut – to jedna z cech (parametrów) opisujących obiekt za pośrednictwem wartości reprezentujących ten atrybut. Wartości te są określonego typu i mogą posiadać wartości z pewnego zakresu lub zbioru.

Cecha diagnostyczna – deskryptor numeryczny opisujący i charakteryzujący analizowany proces, zwany również atrybutem procesu.

Ekstrakcja cech diagnostycznych – to proces tworzenia atrybutów wejściowych dla modelu eksploracji na podstawie wyników pomiarowych.

Proces ten nazywany jest również **generacją cech**. Proces ten może być powiązany z normalizacją, standaryzacją lub inną transformacją danych, mających na celu uwydawnienie głównych cech modelowanego procesu, które mają istotny wpływ na budowę modelu oraz uzyskiwane wyniki i uogólnienie.

Generalizacja – to zdolność lub właściwość modelu eksploracji danych polagająca na możliwości poprawnego działania (np. przewidywania, klasyfikacji, regresji) modelu na innych danych niż dane uczące.

PODSTAWOWE POJĘCIA I TERMINOLOGIA

Grupowanie (klasteryzacja) – to proces wyszukiwania obiektów zdefiniowanych przy pomocy danych podobnych do siebie.

Klasyfikacja – to proces przyporządkowywania obiektów do określonych klas na podstawie podobieństwa lub innych procesów skojarzeniowych albo w wyniku regresji na podstawie analizy i przetwarzania danych wejściowych, której wynikiem jest wartość odpowiadającą klasie lub stopniu podobieństwa do niej.

Model – to zwykle algorytm lub wzór matematyczny połączony z pewną strukturą lub sposobem reprezentacji przetworzonych danych źródłowych, określany w trakcie procesu uczenia, adaptacji lub konstrukcji.

Obserwacja – to zestaw pomiarów tworzących jeden rekord danych (krotkę).

Predykcja – to wynik procesu regresji lub kojarzenia, w którym otrzymujemy odpowiedź w postaci liczbowej lub innego obiektu.

Redukcja – to proces kompresji stratnej polegający na zmniejszeniu wymiaru wektorów lub macierzy obserwacji poprzez eliminację mało reprezentatywnych lub niekompletnych atrybutów albo w wyniku określania pochodnych reprezentatywnych cech (np. PCA, ICA).

PODSTAWOWE POJĘCIA I TERMINOLOGIA

Adaptacja – to polegający na przedstawieniu danych uczących oraz dobraniu, dopasowaniu lub obliczeniu wartości modelu tak, aby dostosował swoje działanie do określonego zbioru, typu i ew. pożądanych wartości wyjściowych danych uczących.

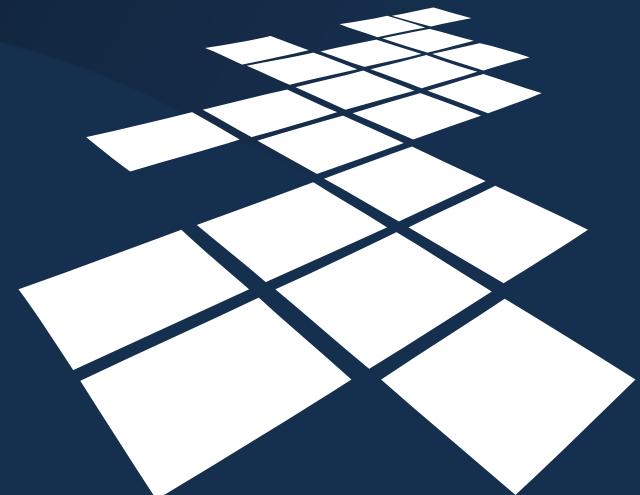
Uczenie – to proces iteracyjny polegający na wielokrotnym przedstawianiu danych uczących oraz poprawianiu wartości modelu tak, aby dostosował swoje działanie do określonego zbioru, typu i ew. pożądanych wartości wyjściowych danych uczących. Uczenie może być np.: nienadzorowane (bez nauczyciela, *unsupervised*), nadzorowane (z nauczycielem, *supervised*), przez wzmacnianie (*reinforcement*), konkurencyjne (*competitive*), motywowane, Bayesowskie, asocjacyjne...

Testowanie – to proces sprawdzania jakości modelu przeprowadzanym w trakcie procesu uczenia lub adaptacji modelu na zbiorze danych chwilowo wydzielonych i wykluczonych z procesu uczenia (tzw. **walidacja** np. krzyżowa – *n-fold cross validation*) lub na zbiorze danych testowych całkowicie wykluczonych z procesu uczenia/adaptacji modelu (testowanie właściwe).

Wzorzec – to zestaw lub sekwencja albo inna struktura danych reprezentowanych w postaci wektora, macierzy, sekwencji albo grafu danych stosowana do budowy, adaptacji, uczenia, walidacji i testowania modelu.

Grupowanie

**Wprowadzanie
Definicja problemu
Klasyfikacja metod
grupowania
Grupowanie hierarchiczne**



**UCZELNIA
ONLINE**

Sformułowanie problemu

Dany jest zbiór obiektów (rekordów).
Znajdź naturalne pogrupowanie obiektów w klasy (klastry, skupienia) obiektów o podobnych cechach

- **Grupowanie**

proces grupowania obiektów, rzeczywistych bądź abstrakcyjnych, w klasy, nazywane klastrami lub skupieniami, o podobnych cechach

Czym jest klaster?

- Istnieje wiele definicji

Zbiór obiektów, które są “podobne”

Zbiór obiektów, takich, że odległość pomiędzy dwoma dowolnymi obiektami należącymi do klastra jest mniejsza aniżeli odległość pomiędzy dowolnym obiektem należącym do klastra i dowolnym obiektem nie należącym do tego klastra

Spójny obszar przestrzeni wielowymiarowej, charakteryzujący się dużą gęstością występowania obiektów

Przykłady (1)

- **Zbiór dokumentów**

Zbiór punktów w przestrzeni wielowymiarowej, w której pojedynczy wymiar odpowiada jednemu słowi z określonego słownika

Współrzędne dokumentu w przestrzeni są zdefiniowane względną częstością występowania słów ze słownika.

Klastry dokumentów odpowiadają grupom dokumentów dotyczących podobnej tematyki

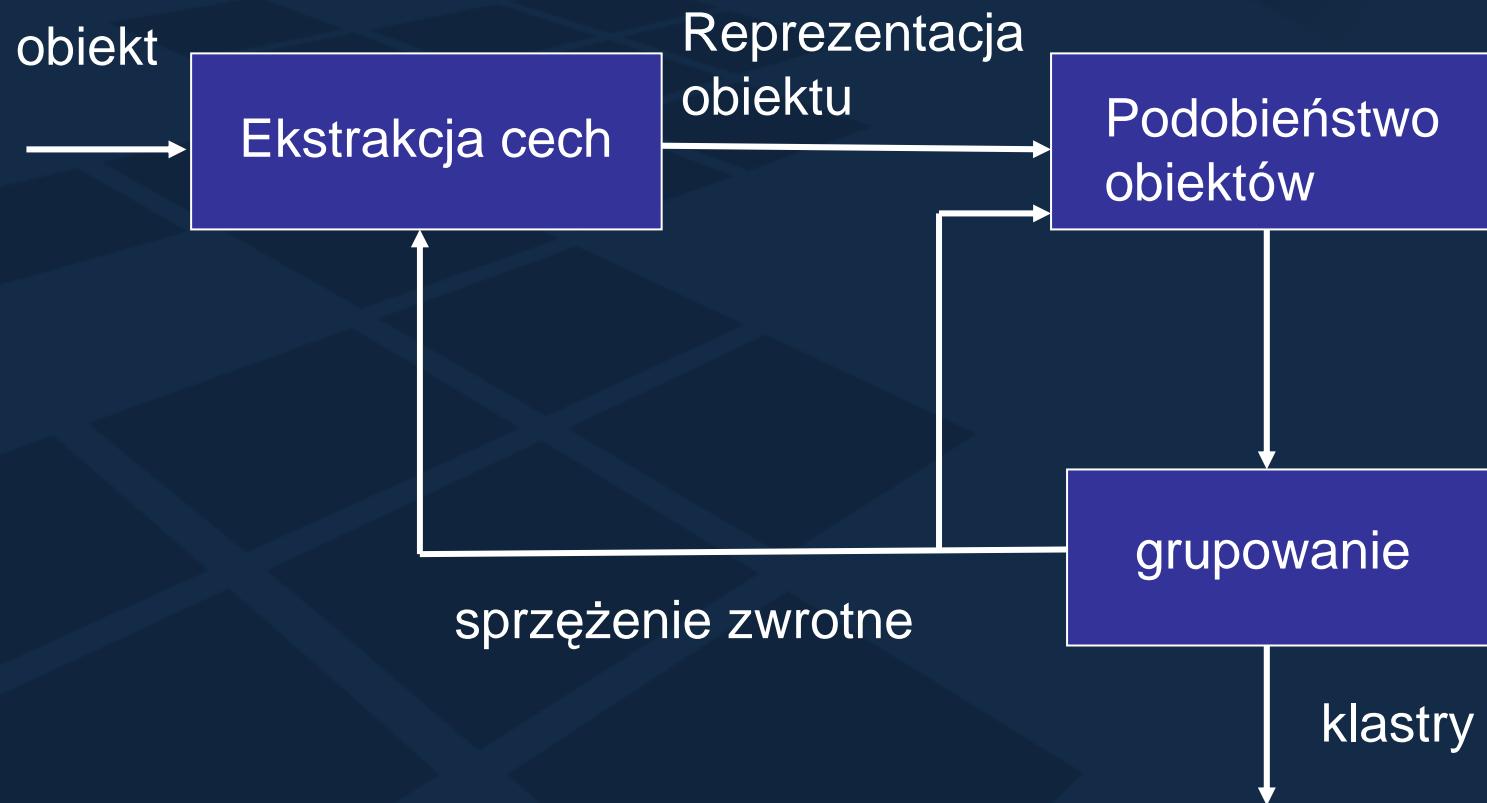
Przykłady (2)

- **Zbiór sekwencji stron WWW**

Pojedyncza sekwencja opisuje sekwencję dostępów do stron WWW danego serwera realizowaną w ramach jednej sesji przez użytkownika

Klastry sekwencji odpowiadają grupom użytkowników danego serwera, którzy realizowali dostęp do tego serwera w podobny sposób

Składowe procesu grupowania (1)



Składowe procesu grupowania (2)

- Proces grupowania

Reprezentacja obiektów
(zawiera ekstrakcję/selekcję cech obiektów)

Definicja miary podobieństwa pomiędzy obiektyami
(zależy od dziedziny zastosowań)

Grupowanie obiektów (klastry)

Znajdowanie charakterystyki klastrów

Miary odległości (1)

- Dyskusja dotycząca podobieństwa, lub odległości, dwóch obiektów wymaga przyjęcia miary odległości pomiędzy dwoma obiektami x i y reprezentowanymi przez punkty w przestrzeni wielowymiarowej
- Klasyczne aksjomaty dla miary odległości

$$1. \ D(x, y) = 0 \Leftrightarrow x = y$$

$$2. \ D(x, y) = D(y, x)$$

$$3. \ D(x, y) \leq D(x, z) + D(z, y) \text{ (nierówność trójkąta)}$$

Miary odległości (2)

- Dana jest k-wymiarowa przestrzeń euklidesowa, odległość pomiędzy dwoma punktami $x=[x_1, x_2, \dots, x_k]$ oraz $y=[y_1, y_2, \dots, y_k]$ można zdefiniować następująco:

Odległość euklidesowa:

(„norma L_2 ”)

$$\sqrt{\sum_{i=1}^k (xi - yi)^2}$$

Odległość Manhattan:

(„miara L_1 ”)

$$\sum_{i=1}^k |xi - yi|$$

Miary odległości (3)

Odległość max z wymiarów:

(„norma L^∞ ”)

$$\max \max_{i=1}^k |x_i - y_i|$$

Odległość Minkowskiego

$$\left(\sum_{i=1}^k (|x_i - y_i|)^q \right)^{1/q}$$

W przypadku, gdy obiekty nie oddają się transformacji do przestrzeni euklidesowej, proces grupowania wymaga zdefiniowania innych miar odległości (podobieństwa): sekwencja dostępów do stron WWW, sekwencje DNA, sekwencje zbiorów, zbiory atrybutów kategorycznych, dokumenty tekstowe, XML, grafy, itp..

Inne miary odległości (1)

- Strony WWW: punkty w przestrzeni wielowymiarowej, w której pojedynczy wymiar odpowiada jednemu słowu z określonego słownika
- Idea: Podobieństwo (odległość) $D(x, y)$ stron x i y zdefiniowane jako iloczyn skalarny wektorów reprezentujących strony x i y . Współrzędne dokumentu w przestrzeni są zdefiniowane jako względna częstotliwość występowania słów ze słownika

Np. przyjmijmy słownik składający się z 4 słów:

$$\begin{array}{lll} x=[2,0,3,1] & y=[5,3,2,0] & x^{\circ}y=16 \\ \text{długość } x=\sqrt{14} & \text{długość } y=\sqrt{38} & \end{array}$$

Inne miary odległości (2)

- Miara odległości

$$D(x, y) = 1 - \frac{x \circ y}{|x| * |y|}$$

- $D(x, y) = 1 - \frac{16}{\sqrt{14}\sqrt{38}} \approx 0.3$

Załóżmy $x = [a_1, a_2, a_3, a_4]$ i $y = [2a_1, 2a_2, 2a_3, 2a_4]$
(strona y jest kopią x), wówczas $D(x, y) = 0$

Inne miary odległości (3)

- Sekwencje DNA, sekwencje dostępu do stron WWW: definicja odległości (podobieństwa) sekwencji symboli, powinna uwzględniać fakt, że sekwencje mogą mieć różną długość oraz różne symbole na tych samych pozycjach, np.: $x = \text{abcde}$ $y = \text{bcdxye}$
- Miara odległości

$$D(x, y) = |x| + |y| - 2 |LCS(x, y)|$$

- gdzie LCS oznacza najdłuższą wspólną podsekwencję (*ang. longest common subsequence*) ($LCS(x, y) = \text{bcde}$). Stąd, $D(x, y) = 3$

Zmienne binarne (1)

- W jaki sposób obliczyć podobieństwo (lub niepodobieństwo) pomiędzy dwoma obiektyami opisanymi zmiennymi binarnymi:
- Podejście: konstruujemy macierz niepodobieństwa

		obiekt j		
		1	0	Sum
obiekt i	1	q	r	q+r
	0	s	t	s+t
	Sum	q+s	r+t	p

q – liczba zmiennych przyjmujących wartość 1 dla obu obiektów
r – ... 1 dla obiektu i, i wartość 0 dla j
s – ... 0 dla obiektu i, i wartość 1 dla j
t – ... 0 dla obu obiektów

$$p=q+r+s+t - \text{łączna liczba zmiennych}$$

Zmienne binarne (2)

- **Zmienne binarne symetryczne**

Zmienną binarną nazywamy symetryczną jeżeli obie wartości tej zmiennej posiadają tą samą wagę (np. płeć).

Niepodobieństwo pomiędzy obiektami i oraz j jest zdefiniowane następująco

$$d(i, j) = \frac{r + s}{q + r + s + t}$$

Zmienne binarne (3)

- **Zmienne binarne asymetryczne**

zmienną binarną nazywamy asymetryczną jeżeli obie wartości tej zmiennej posiadają różne wagi (np. wynik badania EKG)

Niepodobieństwo pomiędzy obiektami i oraz j jest zdefiniowane następująco:

$$d(i, j) = \frac{r + s}{q + r + s}$$

Zmienne binarne (4)

- Dana jest tablica zawierająca informacje o pacjentach

imię	ból	gorączka	katar	test1	test2	test3	test4
Jack	Y	Y	N	P	N	N	N
Mary	N	Y	N	P	N	P	N
Jim	Y	Y	Y	N	N	N	N
...

$$d_{asym}(jack,mary) = \frac{2}{4} = 0.5$$

$$d_{sym}(jack,mary) = \frac{2}{7} = 0.29$$

$$d_{asym}(jack, jim) = \frac{2}{4} = 0.5$$

$$d_{sym}(jack, jim) = \frac{2}{7} = 0.29$$

$$d_{asym}(jim,mary) = \frac{4}{5} = 0.8$$

$$d_{sym}(jim,mary) = \frac{4}{7} = 0.57$$

Zmienne kategoryczne

- **Zmienna kategoryczna** jest generalizacją zmiennej binarnej: może przyjmować więcej niż dwie wartości (np. dochód: wysoki, średni, niski)
- Niepodobieństwo (podobieństwo) pomiędzy obiekty i, j, opisanymi zmiennymi kategorycznymi, można zdefiniować następująco:

$$d(i, j) = \frac{p - m}{p} \quad d(i, j) = \frac{p - n}{p} = d(i, j) = \frac{m}{p}$$

- gdzie p oznacza łączną liczbę zmiennych, m oznacza liczbę zmiennych, których wartość jest identyczna dla obu obiektów, n oznacza liczbę zmiennych, których wartość jest różna dla obu obiektów

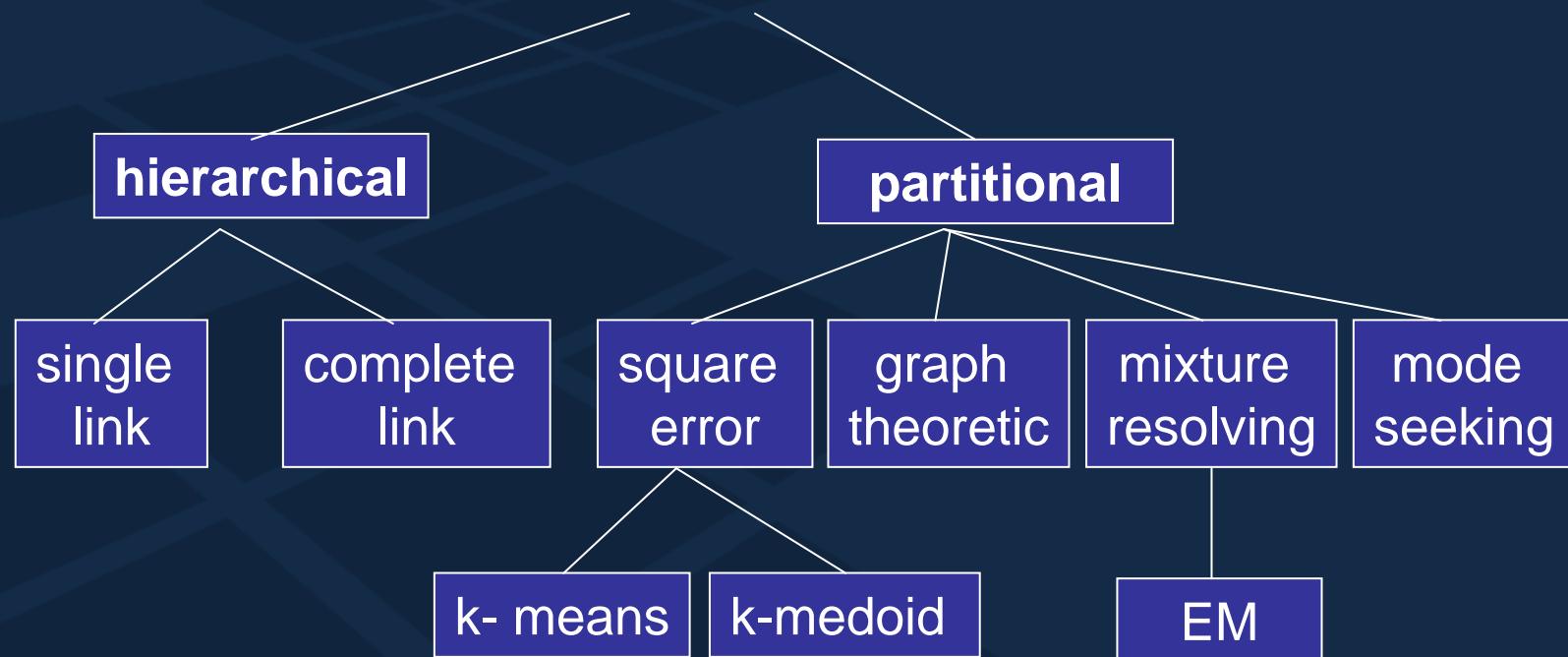
Metody grupowania

Typy metod

- Istnieje wiele różnych metod i algorytmów grupowania:
 - Dla danych liczbowych i/lub danych symbolicznych
 - Deterministyczne i probabilistyczne
 - Rozłączne i przecinające się
 - Hierarchiczne i płaskie
 - Monoteiczny i politeiczny
 - Przyrostowe i nieprzyrostowe

Metody grupowania (1)

Klasyfikacja metod



Metody grupowania (4)

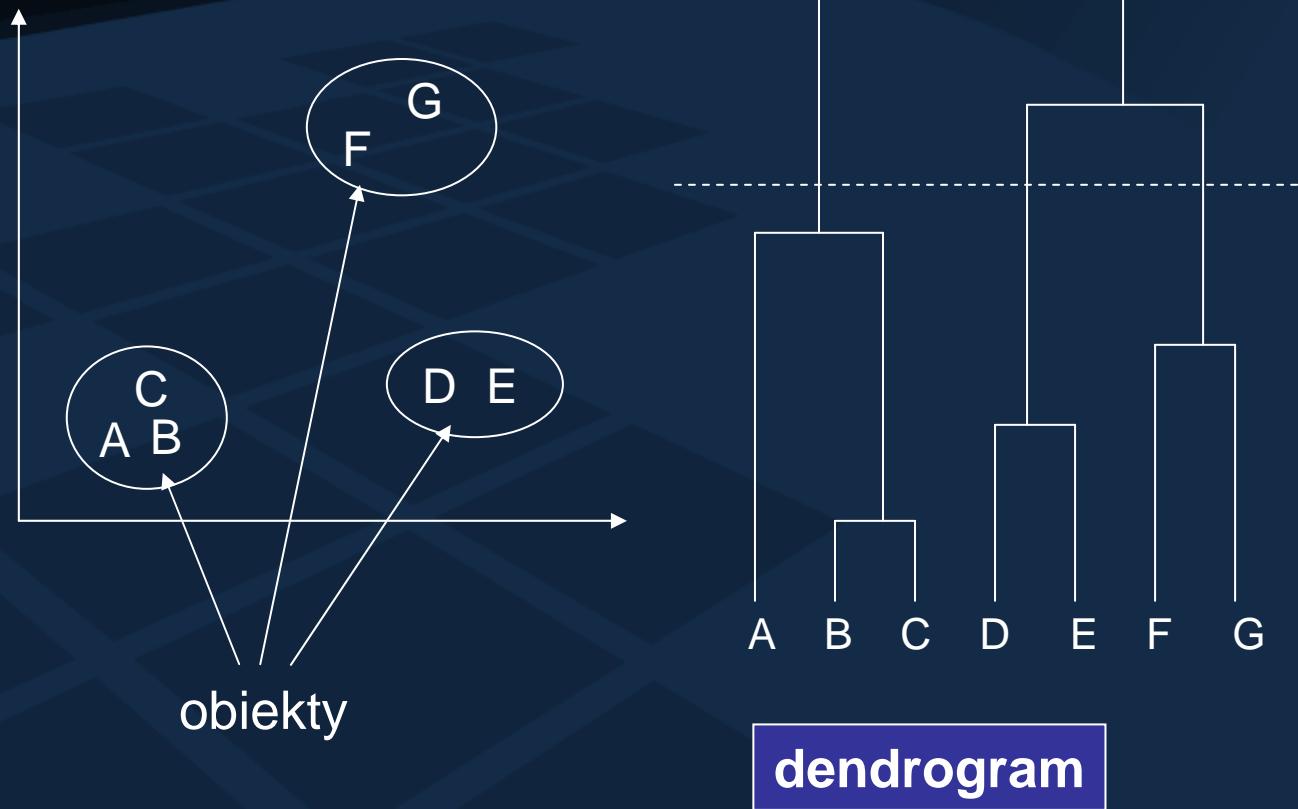
- Dwa podstawowe podejścia do procesu grupowania obiektów:
- **Metody hierarchiczne**

generują zagnieżdżoną sekwencję podziałów zbiorów obiektów w procesie grupowania

- **Metody z iteracyjno-optymalizacyjne**

generują tylko jeden podział (partycję) zbioru obiektów w dowolnym momencie procesu grupowania

Metody grupowania hierarchicznego (1)



Metoda grupowania hierarchicznego polega na sekwencyjnym grupowaniu obiektów – drzewo klastrów (tzw. dendrogram)

Metody grupowania hierarchicznego (2)

- **Podejście podziałowe**

(top-down): początkowo, wszystkie obiekty przypisujemy do jednego klastra; następnie, w kolejnych iteracjach, klaster jest dzielony na mniejsze klastry, które, z kolei, dzielone są na kolejne mniejsze klastry

- **Podejście aglomeracyjne**

(bottom-up): początkowo, każdy obiekt stanowi osobny klaster, następnie, w kolejnych iteracjach, klastry są łączone w większe klastry

Miary odległości (1)

- W obu podejściach, aglomeracyjnym i podziałowym, liczba klastrów jest ustalona z góry przez użytkownika i stanowi warunek stopu procesu grupowania

4 podstawowe (najczęściej stosowane) miary odległości pomiędzy klastrami są zdefiniowane następująco, gdzie $|p - p'|$ oznacza odległość pomiędzy dwoma obiektami (lub punktami) p i p' , m_i oznacza średnią wartość klastra C_i , i n_i oznacza liczbę obiektów należących do klastra C_i

Miary odległości (2)

Minimalna odległość:

$$d_{\min}(C_i, C_j) = \min_{p \in C_i, p' \in C_j} \|p - p'\|$$

Maksymalna odległość:

$$d_{\max}(C_i, C_j) = \max_{p \in C_i, p' \in C_j} \|p - p'\|$$

Miary odległości (3)

Odległość średnich:

$$d_{mean}(C_i, C_j) = \|m_i - m_j\|$$

Średnia odległość:

$$d_{ave}(C_i, C_j) = 1 / (n_i n_j) \sum_{p \in C_i} \sum_{p' \in C_j} \|p - p'\|$$

Ogólny hierarchiczny aglomeracyjny algorytm grupowania

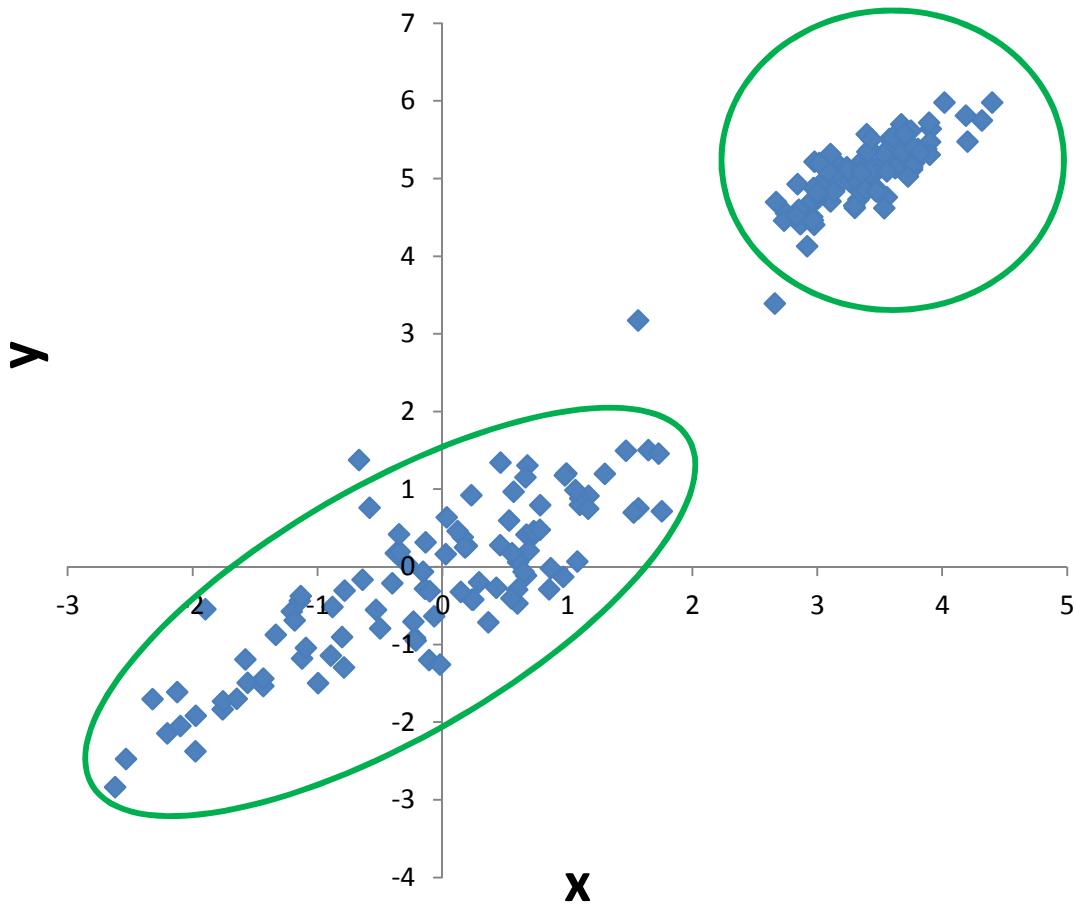
- Dane wejściowe: baza danych D obiektów (n- obiektów)
- Dane wyjściowe: dendrogram reprezentujący grupowanie obiektów
 - 1) umieść każdy obiekt w osobnym klastrze;
 - 2) skonstruuj macierz odległości pomiędzy klastrami;
 - 3) Dla każdej wartości niepodobieństwa dk
(dk może się zmieniać w kolejnych iteracjach)
powtarzaj
 - Utwórz graf klastrów, w którym każda para klastrów, której wzajemna odległość jest mniejsza niż dk, jest połączona lukiem aż wszystkie klastry utworzą graf spójny

Hierarchiczny aglomeracyjny algorytm grupowania

- Umieść każdy obiekt w osobnym klastrze.
Skonstruuj macierz przyległości zawierającą odległości pomiędzy każdą parą klastrów
- Korzystając z macierzy przyległości znajdź najbliższą parę klastrów. Połącz znalezione klastry tworząc nowy klaster. Uaktualnij macierz przyległości po operacji połączenia
- Jeżeli wszystkie obiekty należą do jednego klastra, zakończ procedurę grupowania, w przeciwnym razie przejdź do kroku 2

Grupowanie

Grupowanie



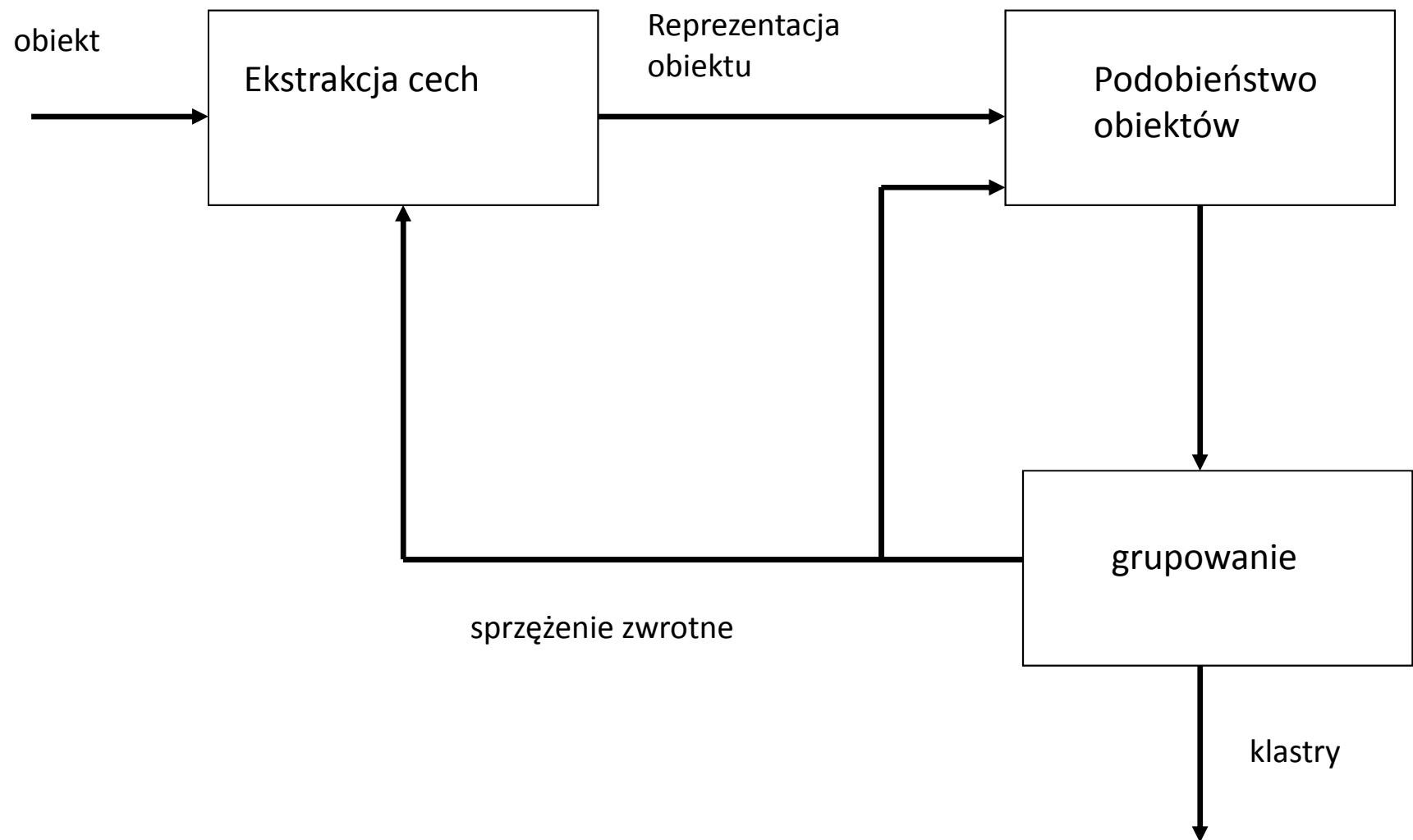
Wprowadzenie

- Celem procesu grupowania jest podział zbioru obiektów, fizycznych lub abstrakcyjnych, na klasy obiektów o podobnych cechach, nazywane klastrami lub skupieniami

Alternatywne definicje

- Zbiór obiektów, które są “podobne”
- Zbiór obiektów, takich, że odległość pomiędzy dwoma dowolnymi obiektemi należącymi do klastra jest mniejsza aniżeli odległość pomiędzy dowolnym obiektem należącym do klastra i dowolnym obiektem nie należącym do tego klastra
- Spójny obszar przestrzeni wielowymiarowej, charakteryzujący się dużą gęstością występowania obiektów

Proces grupowania



Niepodobieństwo obiektów

- Niepodobieństwo (podobieństwo) obiektów opisujemy za pomocą macierzy niepodobieństwa (podobieństwa)
- Danych jest N obiektów, z których każdy jest opisany wartościami p atrybutów A1, ..., Ap (nazywanych zmiennymi)
- Macierz niepodobieństwa obiektów D, typu $N \times N$ opisuje niepodobieństwo pomiędzy każdą parą obiektów:

$$D = \begin{bmatrix} 0 & D(x_1, x_2) & \dots & D(x_1, x_N) \\ D(x_2, x_1) & 0 & \dots & D(x_2, x_N) \\ \vdots & \vdots & 0 & \vdots \\ D(x_N, x_1) & D(x_N, x_2) & \dots & 0 \end{bmatrix}$$

- Gdzie $D(x_i, x_j)$ oznacza niepodobieństwo obiektów

Odległość pomiędzy obiektami

- Konieczne jest znalezienie miary odległości
- Bardzo często używane miary odległości są metrykami:
 - $D(x, y) > 0$
 - $x=y \iff D(x, y) = 0$
 - $D(x, y) = D(y, x)$
 - $D(x, y) \leq D(x, z) + D(z, y)$ (nierówność trójkąta)

Metryka Minkowskiego

$$d(X, Y) = \left(\sum_{i=1}^n |x_i - y_i|^h \right)^{1/h}$$

Odległość ze
względu na i-ty
atrybut
 $d(x_i, y_i)$

- Specjalne przypadki
 - Manhattan: $h = 1$
 - Euklidesowa: $h = 2$
 - Chebysheva: $h = \infty$

$$\text{cheb}(X, Y) = \max|x_i - y_i|$$

Atrybuty kategoryczne

- Odległość Jaccarda
- (ew. znormalizowana) odległość Hamminga
- Macierz niepodobieństwa:

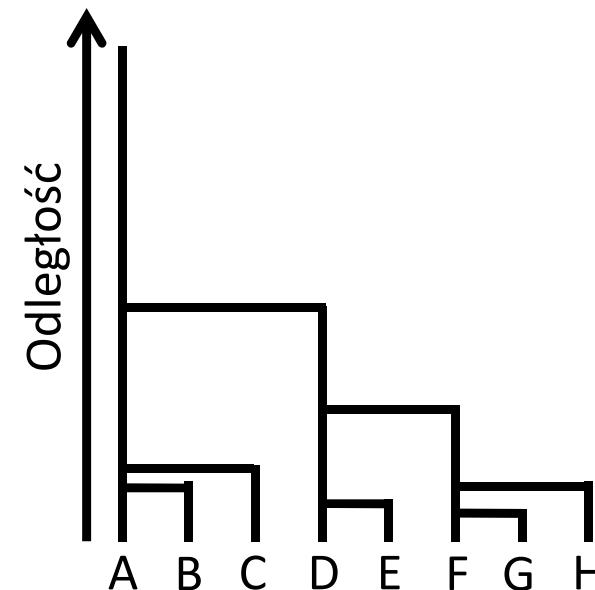
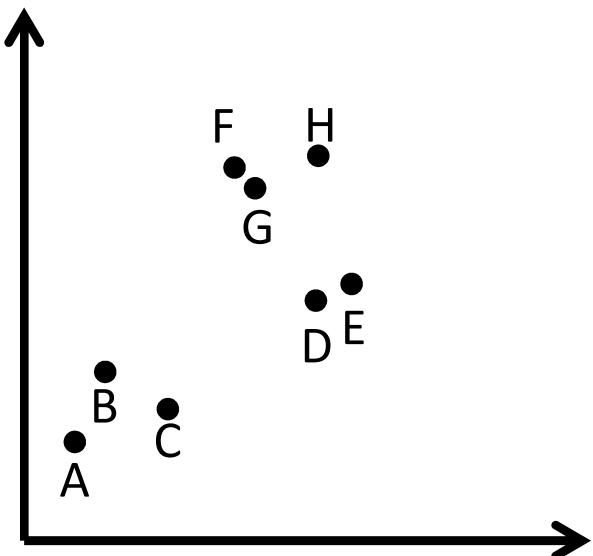
	Honda	Fiat	Volvo	Nissan
Honda	0	1	1	0
Fiat	1	0	1	1
Volvo	1	1	0	1
Nissan	0	1	1	0

Klasyfikacja metod grupowania

- metody hierarchiczne
- metody iteracyjno-optymalizacyjne
- metody gęstościowe
- metody gridowe

Metody hierarchiczne

Aglomeracyjne/Podziałowe



Metody hierarchiczne

- Metoda grupowania hierarchicznego polega na sekwencyjnym grupowaniu obiektów - drzewo klastrów (tzw. dendrogram).
 - Podejście podziałowe (top-down): początkowo, wszystkie obiekty przypisujemy do jednego klastra; następnie, w kolejnych iteracjach, klaster jest dzielony na mniejsze klastry, które, z kolei, dzielone są na kolejne mniejsze klastry
 - Podejście aglomeracyjne (bottom-up): początkowo, każdy obiekt stanowi osobny klaster, następnie, w kolejnych iteracjach, klastry są łączone w większe klastry

Odległość pomiędzy zgrupowaniami

- Oznaczenia:
 - $D(x_i, x_j)$ – odległość pomiędzy obiektami x_i i x_j
 - m_i – średnia wartość zgrupowania C_i
 - n_i – liczba obiektów w zgrupowaniu C_i

Odległość pomiędzy zgrupowaniami

- Minimalna odległość:

$$d_{min} = \min_{x_i \in C_i, x_j \in C_j} D(x_i, x_j)$$

- Maksymalna odległość:

$$d_{max} = \max_{x_i \in C_i, x_j \in C_j} D(x_i, x_j)$$

- Średnia odległość

$$d_{avg} = \frac{1}{n_i n_j} \sum_{x_i \in C_i} \sum_{x_j \in C_j} D(x_i, x_j)$$

- Odległość średnich

$$d_{mean} = D(m_i, m_j)$$

Hierarchiczny algorytm grupowania

- Wejście: baza danych D n obiektów.
 - Wyjście: dendrogram reprezentujący sekwencję grupowania obiektów
1. umieść każdy obiekt w osobnym klastrze;
 2. skonstruuj macierz odległości międzyklastrowej dla wszystkich par klastrów;
 3. korzystając z macierzy odległości międzyklastrowych, znajdź najbliższą parę klastrów i połącz znalezione klastry, tworząc nowy klaster;
 4. uaktualnij macierz odległości międzyklastrowych po operacji połączenia;

Hierarchiczny algorytm grupowania

5. if wszystkie obiekty należą do jednego klastra then
6. zakończ procedurę grupowania;
7. else
8. przejdź do kroku 3;
9. end if
10. return dendrogram reprezentujący sekwencje grupowania obiektów;

Alg. iteracyjno-optymalizacyjne

- Dane: k – liczba zgrupowań
- Utwórz wstępny podział na k zgrupowań
- W kolejnych iteracjach przesuwaj obiekty pomiędzy zgrupowaniami tak długo jak poprawia to jakość grupowania.
- Optymalne rozwiązanie można osiągnąć jedynie po przeanalizowaniu wszystkich możliwych podziałów.
- W praktyce stosuje się jedną z wielu technik optymalizacji kombinatorycznej: iterative improvement, tabu search, simulating annealing, genetic algorithms, ant algorithms, itp

Miary oceny zgrupowania

- Dwa aspekty grupowania:
 - Klastry powinny być zwarte
 - Klastry powinny być maksymalnie rozłączne
- Odchylenie wewnątrzklastrowe - $wc(C)$
- Odchylenie międzyklastrowe – $bc(C)$
- Średnia klastra (mean) - m_k
$$m_k = \frac{1}{n_K} \sum_{x \in C_k} x$$
 - Gdzie:
 - n_K - liczba obiektów należących do k-tego klastra C_k

Miary oceny zgrupowania

- Miary $wc(C)$:

$$wc(C) = \sum_{i=1}^k wc(C_i) = \sum_{i=1}^k \sum_{x_j \in Ci} d(x_j, m_i)^2$$

$$wc(C) = \max_i \min_{y_j \in C_k} \{d(x_i, y_j) | x_i \in C_k, x_i \neq y_j\}$$

- Miara $bc(C)$:

$$bc(C) = \sum_{1 \leq i < j \leq k} d(r_i, r_j)^2$$

Algorytm k-średnich

- Algorytm zachłanny
- Optymalizuje miarę:

$$wc(C) = \sum_{i=1}^k wc(C_i) = \sum_{i=1}^k \sum_{x_j \in Ci} d(x_j, m_i)^2$$

– Gdzie

- $m_k = \frac{1}{n_k} \sum_{x \in C_k} x$
- Zgrupowanie jest reprezentowane przez średnią
- Obiekt należy do tego zgrupowania, którego środek jest najbliżej

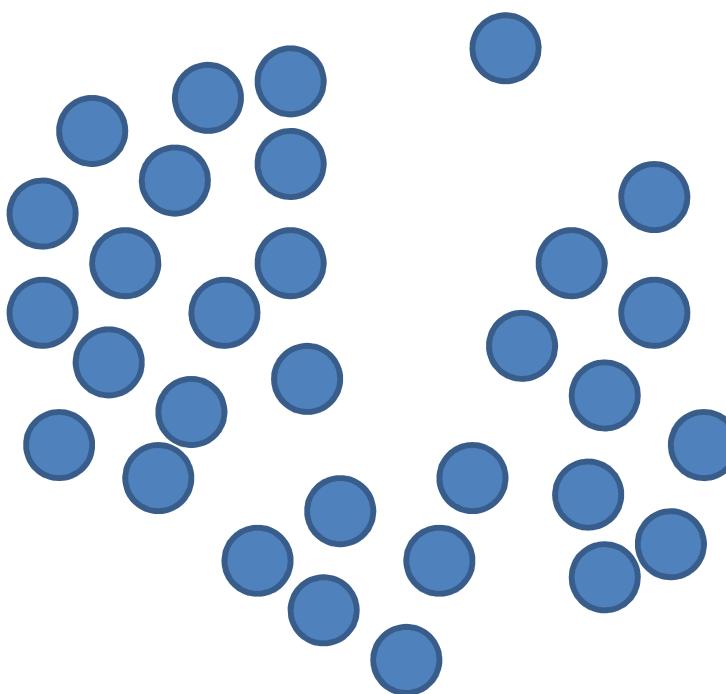
Algorytm k-średnich

Wejście: liczba zgrupowań k , baza danych n obiektów

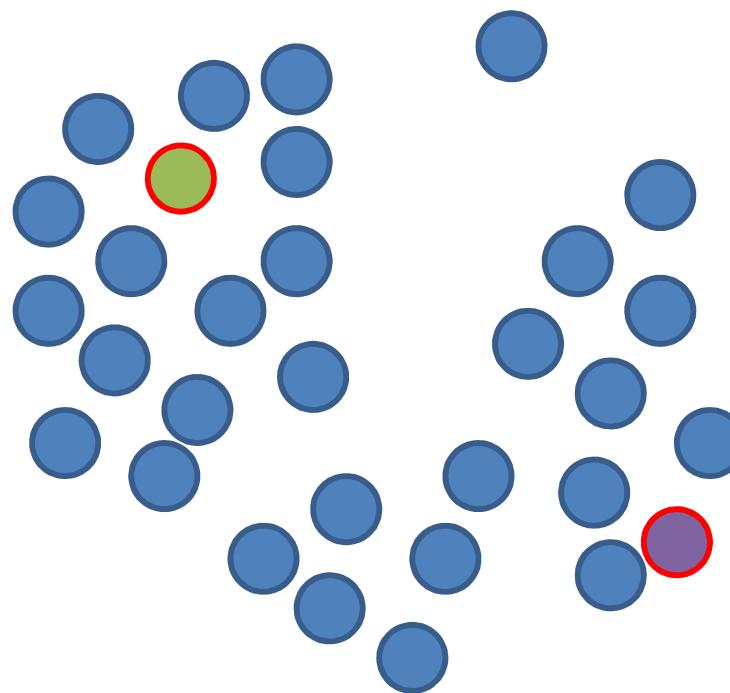
Wyjście: zbiór k zgrupowań minimalizujący kryterium błędu średniokwadratowego

1. wybierz losowo k obiektów, jako środki skupień
2. while przydział obiektów się zmienił do
3. każdy obiekt przydziel do zgrupowania, którego środek jest najbliżej
4. oblicz nowe środki zgrupowań - m_k
5. end while

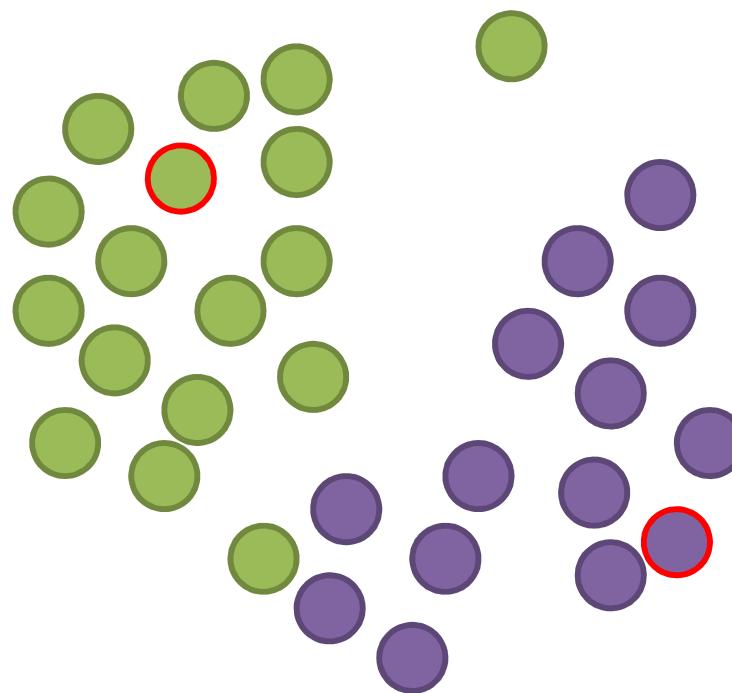
Grupowanie - KMeans



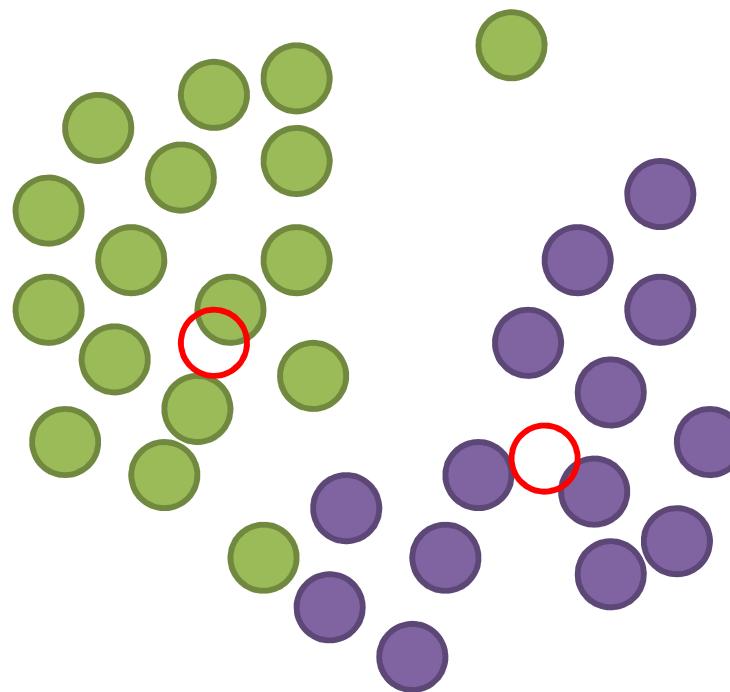
Grupowanie - KMeans



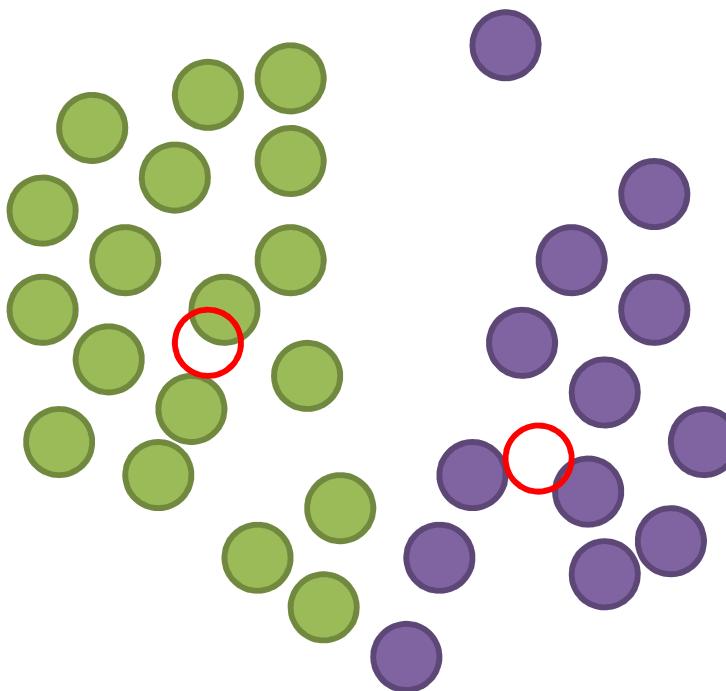
Grupowanie - KMeans



Grupowanie - KMeans



Grupowanie - KMeans



Problem początkowego podziału

- Wynik działania algorytmu (tj. ostateczny podział obiektów pomiędzy klastrami) silnie zależy od początkowego podziału obiektów
- Algorytm łatwo wpada w minimum lokalne
- W celu zwiększenia szansy znalezienia optimum globalnego należy kilkakrotnie uruchomić algorytm dla różnych podziałów początkowych

Problem punktów osobliwych

- Wadą algorytmu k-srednich jest jego czułość na występowanie punktów osobliwych
- Punktami osobliwymi nazywamy obiekty, które znaczco różnią się od pozostałych grupowanych obiektów
- Punkty osobliwe reprezentują najczęściej dwa przypadki:
 - błędy w danych
 - obiekty o bardzo specyficznych wartościach atrybutów
- Najprostsze rozwiązanie polega na usunięciu obiektów, które znaczco odbiegają od środków klastrów

Przykład 1

- Dane są punkty:

A = (2, 1)	B = (4, 2)	C = (4, 3)	D = (10, 5)
E = (11, 5)	F = (9, 9)	G = (12, 9)	

- Przyjmujemy k=3.
- Jako początkowe środki przyjmujemy współrzędne punktów A, B i C

Przykład 1

Odległości

	A	B	C	D	E	F	G
(2,1)	0	2,24	2,83	8,94	9,85	10,63	12,81
(4,2)	2,24	0	1	6,71	7,62	8,6	10,63
(4,3)	2,838	1	0	6,32	7,28	7,81	10

Nowe środki

$$\frac{1}{1} * (2,1) = (2,1)$$

$$\frac{1}{1} * (4,2) = (4,2)$$

$$\frac{1}{5} * ((4,3) + (10,5) + (11,5) + (9,9) + (12,9)) = \frac{1}{5} * (46,31) = (9.2,6.2)$$

Przykład 1

Odległości

	A	B	C	D	E	F	G
(2,1)	0	2,24	2,83	8,94	9,85	10,63	12,81
(4,2)	2,24	0	1	6,71	7,62	8,6	10,63
(9,2,6,2)	8,88	6,68	6,11	1,44	2,16	2,81	3,96

Nowe środki

$$\frac{1}{2} * ((2,1) + (4,3)) = (3,2)$$

$$\frac{1}{1} * (4,2) = (4,2)$$

$$\frac{1}{4} * ((10,5) + (11,5) + (9,9) + (12,9)) = \frac{1}{4} * (42,28) = (10,5,7)$$

Przykład 1

Odległości

	A	B	C	D	E	F	G
(3,2)	1,41	1	1,41	7,62	8,54	9,22	11,4
(4,2)	2,24	0	1	6,71	7,62	8,6	10,63
(10,5,7)	10,4	8,2	7,63	2,06	2,06	2,5	2,5

Nowe środki

$$\frac{1}{1} * (2,1) = (2,1)$$

$$\frac{1}{2} * ((4,2) + (4,3)) = (4,2.5)$$

$$\frac{1}{4} * ((10,5) + (11,5) + (9,9) + (12,9)) = \frac{1}{4} * (42,28) = (10.5,7)$$

Przykład 1

Odległości

	A	B	C	D	E	F	G
(2,1)	0	2,24	2,83	8,94	9,85	10,63	12,81
(4,2.5)	2,5	0,5	0,5	6,5	7,43	8,2	10,31
(10.5,7)	10,4	8,2	7,63	2,06	2,06	2,5	2,5

Koniec, ponieważ przypisanie obiektów do zgrupowań się nie zmieniło

Przykład 2

- Wykonamy teraz przebieg dla innego początkowego podziału.
- Jako początkowe środki przyjmujemy współrzędne punktów A, D i F

Przykład 2

Odległości

	A	B	C	D	E	F	G
(2,1)	0	2,24	2,83	8,94	9,85	10,63	12,81
(10,5)	8,94	6,71	6,32	0	1	4,12	4,47
(9,9)	10,63	8,6	7,81	4,12	4,47	0	3

Nowe środki

$$\frac{1}{3} * ((2,1) + (4,2) + (4,3)) = \left(\frac{10}{3}, 3\right)$$

$$\frac{1}{2} * ((10,5) + (11,5)) = (10.5, 5)$$

$$\frac{1}{2} * ((9,9) + (12,9)) = \frac{1}{2} * (21,18) = (10.5, 9)$$

Przykład 2

Odległości

	A	B	C	D	E	F	G
(3.33,3)	2,4	1,2	0,67	6,96	7,92	8,25	10,54
(10.5,5)	9,39	7,16	6,8	0,5	0,5	4,27	4,27
(10.5,9)	11,67	9,55	8,85	4,03	4,03	1,5	1,5

Koniec, ponieważ przypisanie obiektów do zgrupowań się nie zmieniło

Przykład

- Porównajmy jakość jednego i drugiego grupowania:
 - Przykład 1
 - Zgrupowanie 1: (2,1) - A
 - Zgrupowanie 2: (4,2.5) – B, C
 - Zgrupowanie 3: (10.5,7) – D, E, F, G
 - $wc(C) = 2 * \frac{1^2}{2} + 2 * 2.06^2 + 2 * 2.5^2 \approx 21.49$
 - Przykład 2
 - Zgrupowanie 1: (3.333,2) – A, B, C
 - Zgrupowanie 2: (10.5,5) – D, E
 - Zgrupowanie 3: (10.5,9) – F, G
 - $wc(C) = 2.4^2 + 1.2^2 + 0.67^2 + 2 * \frac{1^2}{2} + 2 * 4.27^2 \approx 12.65$

Grupowanie gęstościowe

- Poprzednie metody grupowania tworzą „sferyczne zgrupowania”
- Proces grupowania w metodach grupowania gęstościowego jest oparty na pojęciu *gęstości* (ang. *density*)
- Zgrupowaniem obiektów jest obszar w przestrzeni obiektów charakteryzujący się dużą gęstością obiektów.
- Zgrupowania obiektów są odseparowane od siebie obszarami o małej gęstości obiektów

Podstawowa idea

- Wybierz 1 obiekt jako „zalążek” zgrupowania
- Dokładaj do niego kolejne obiekty tak długo jak dookoła jest wystarczające zagęszczenie obiektów
- Wybierz kolejny obiekt jeszcze nie przydzielony jako kolejny „zalążek” i... patrz poprzedni punkt ☺
- Postępuj w ten sposób tak długo aż wszystkie obiekty zostaną przydzielone

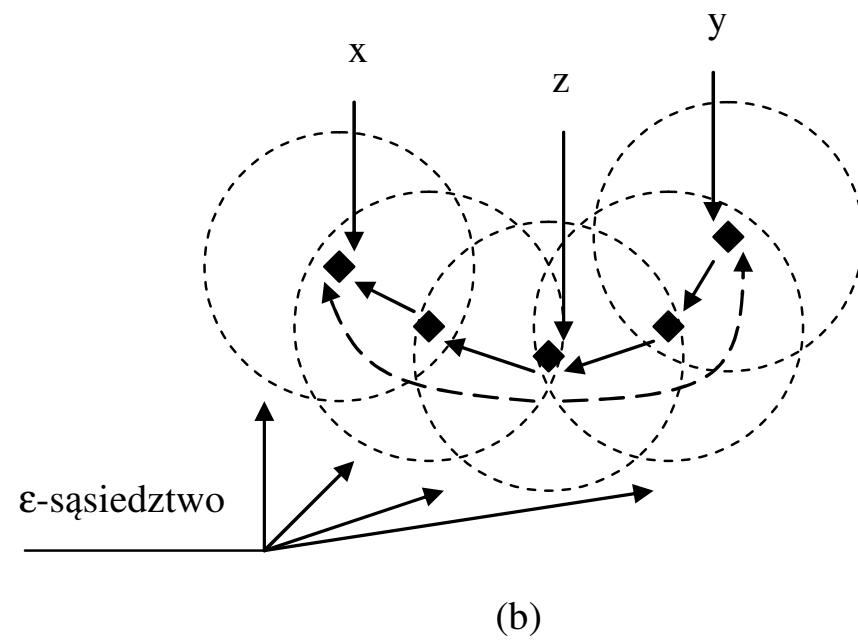
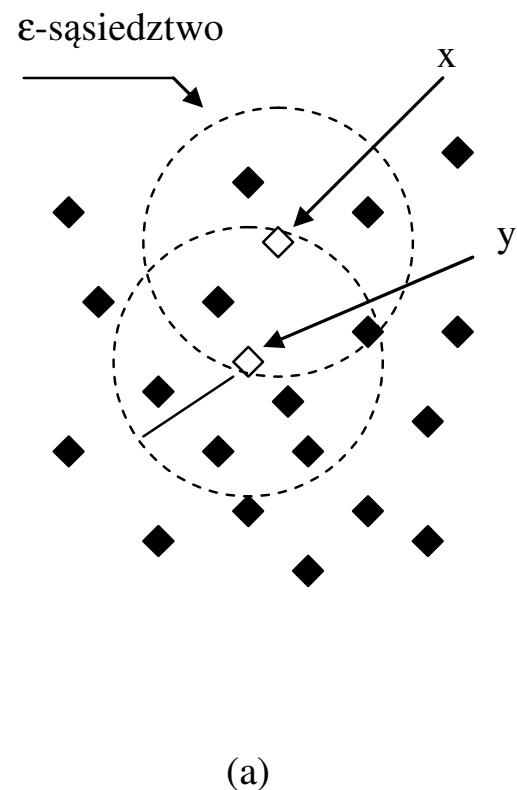
Podstawowe terminy

- Oznaczenia:
 - D – zbiór grupowanych obiektów
 - $D(x_i, x_j)$ – odległość pomiędzy obiektemi x_i i x_j
- $N_\varepsilon(x)$ – ε – sąsiedztwo obiektu x – zbiór obiektów nie dalej niż ε od obiektu x :
$$N_\varepsilon(x) = \{y \in D | D(x, y) \leq \varepsilon\}$$
- Obiekt *centralny* – obiekt, którego ε – sąsiedztwo jest większe niż minPts

Podstawowe terminy

- Mówimy, że obiekt x jest bezpośrednio gęstościowo osiągalny z obiektu y , jeżeli:
 - $x \in N_\varepsilon(y)$
 - Y jest obiektem centralnym
- Mówimy, że obiekt x jest gęstościowo osiągalny z obiektu y , jeżeli istnieje łańcuch obiektów p_1, p_2, \dots, p_n , że:
 - $p_1=y, p_n=x$
 - p_{i+1} jest bezpośrednio gęstościowo osiągalny z p_i
 $(1 \leq i < n)$

Przykład



Podstawowe terminy

- Obiekt x jest *gęstościowo połączony* z obiektem y , względem parametrów ϵ i MinPts , jeżeli istnieje obiekt $z \in D$, taki że oba obiekty x i y są gęstościowo osiągalne z obiektu z względem parametrów ϵ i MinPts

Zgrupowanie

- Zbiór $C \subseteq D$ taki, że:
 - dla dowolnych obiektów $x, y \in D$: jeżeli $x \in C$ i y jest gęstościowo osiągalny z obiektu x , względem parametrów ε i MinPts , wówczas $y \in C$
 - dla dowolnych obiektów $x, y \in C$: obiekt x jest gęstościowo połączony z obiektem y , względem parametrów ε i MinPts

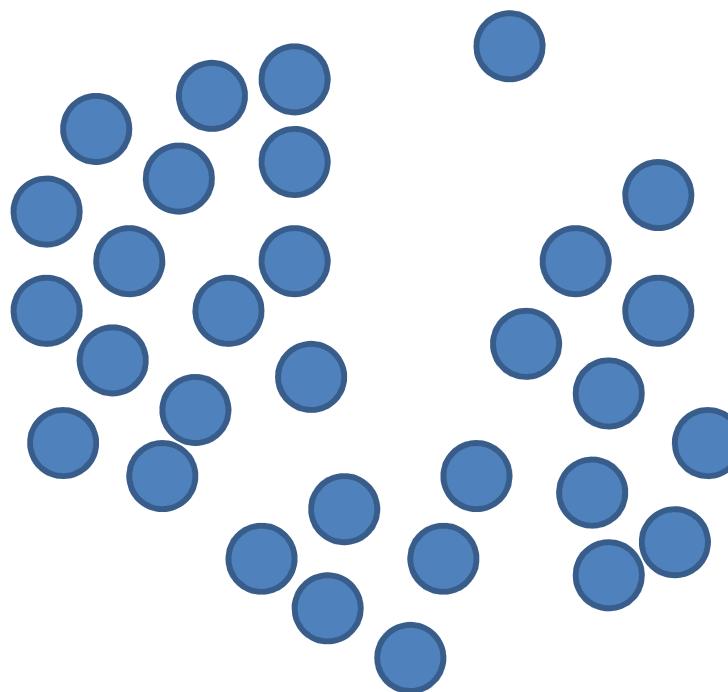
Oznaczenia

- Niech C_1, \dots, C_k oznacza zbiór klastrów zbioru obiektów D względem parametrów ϵ i MinPts
- Zbiorem punktów osobliwych jest taki podzbiór obiektów zbioru D, który nie należy do żadnego klastra C_i , $i = 1, \dots, k$
- Obiekty nie należące do żadnego zgrupowania to obiekty osobliwe

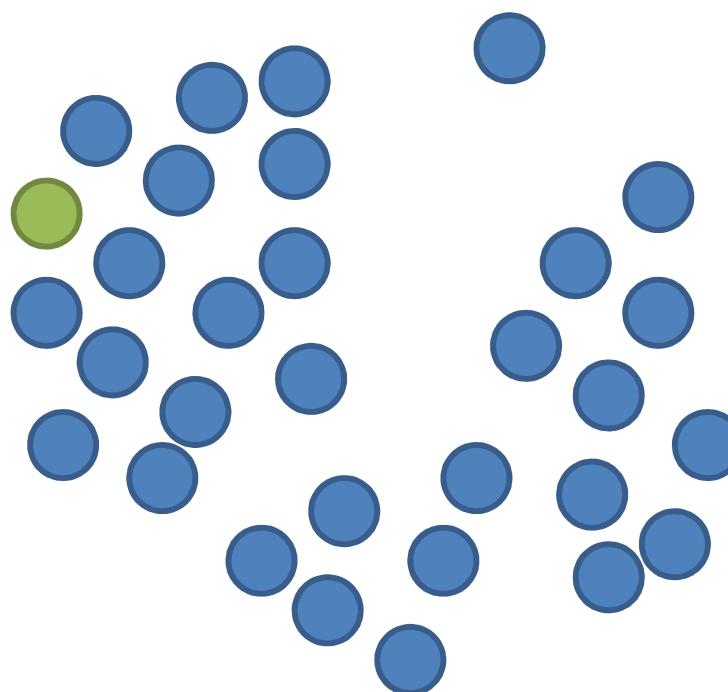
Algorytm DBSCAN

- Algorytm jest realizowany w trzech krokach:
 1. rozpoczęamy od dowolnego obiektu $x \in D$,
 2. jeżeli ϵ - sąsiedztwo obiektu p spełnia warunek minimalnej gęstości, to jest $|N_\epsilon(x)| \geq \text{MinPts}$, wówczas tworzony jest klaster C i wszystkie obiekty gęstościowo osiągalne z obiektu x są dołączane do klastra C ; w przeciwnym razie (obiekt x nie jest obiektem centralnym), wracamy do kroku (1) i wybieramy następny obiekt zbioru D ,
 3. proces grupowania jest kontynuowany tak długo, aż zostaną przetworzone wszystkie obiekty zbioru D
- Obiekty, które nie zostały zaklasyfikowane do żadnego z klastrów, tworzą zbiór punktów osobliwych

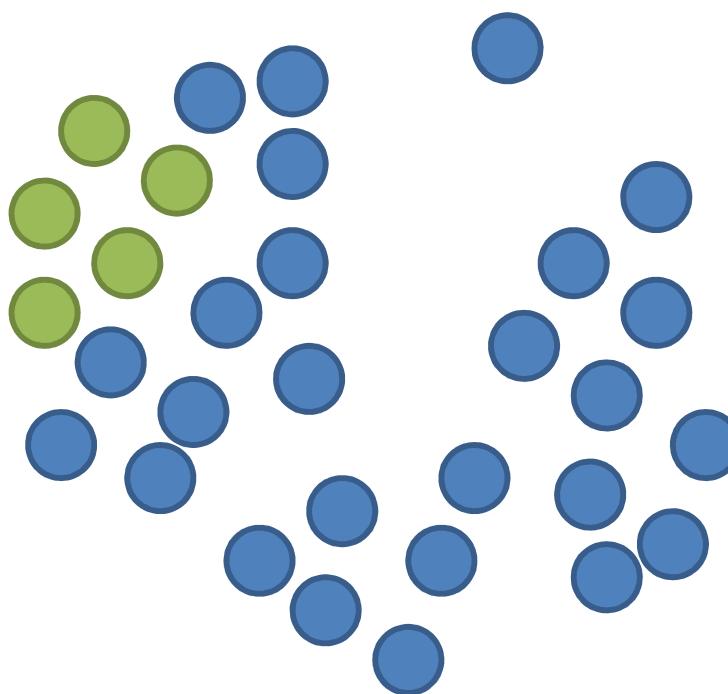
Grupowanie - DBSCAN



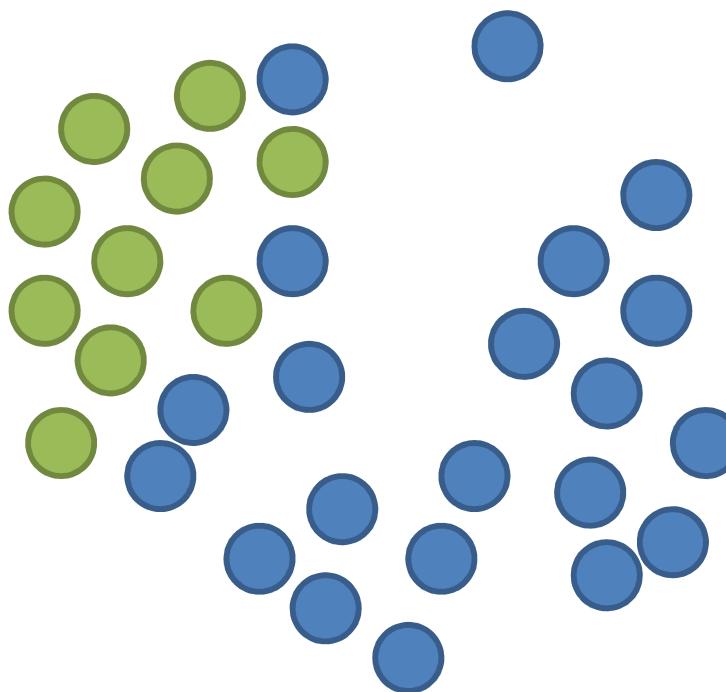
Grupowanie - DBSCAN



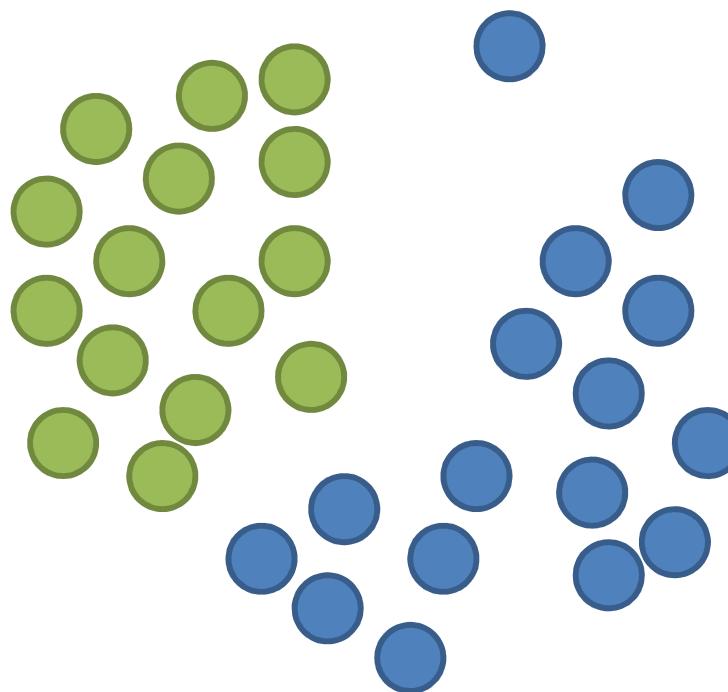
Grupowanie - DBSCAN



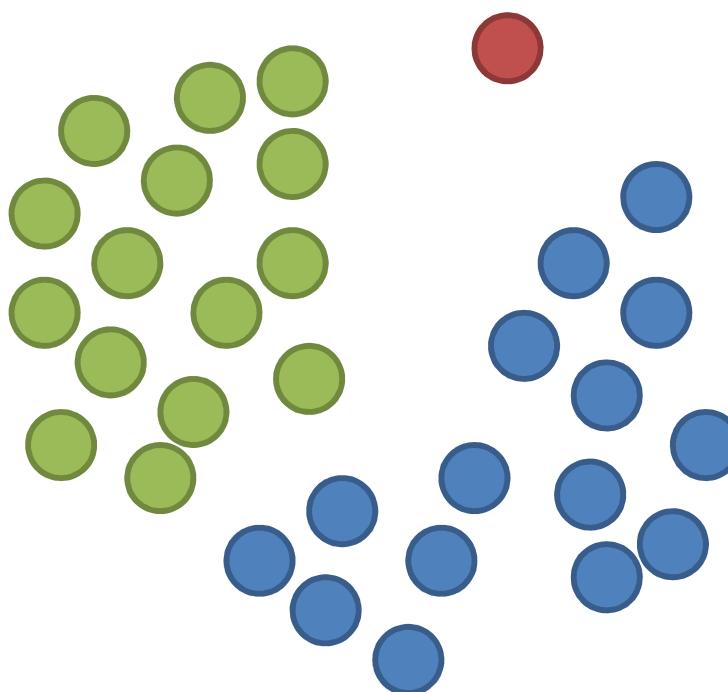
Grupowanie - DBSCAN



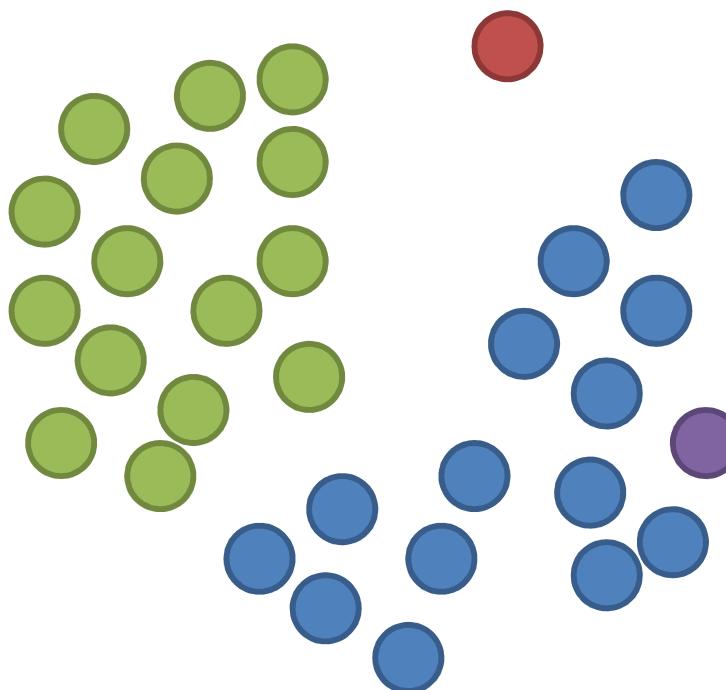
Grupowanie - DBSCAN



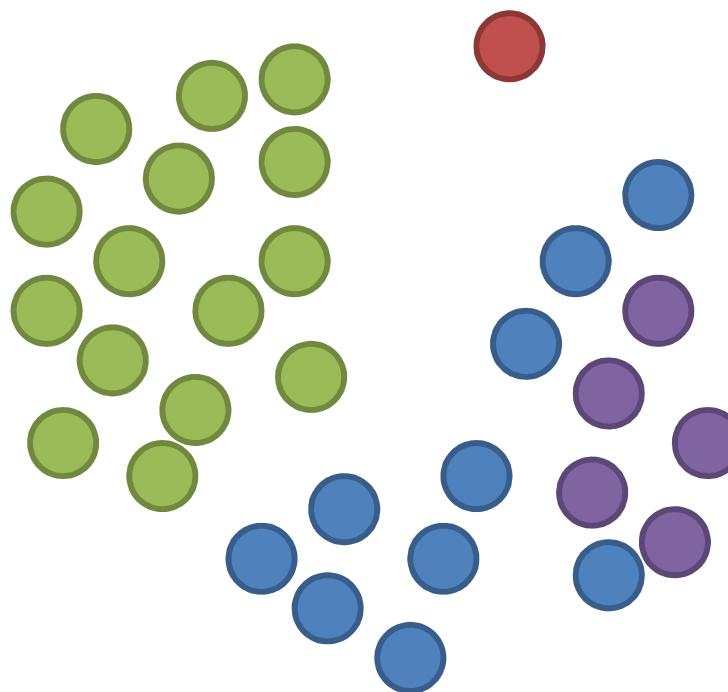
Grupowanie - DBSCAN



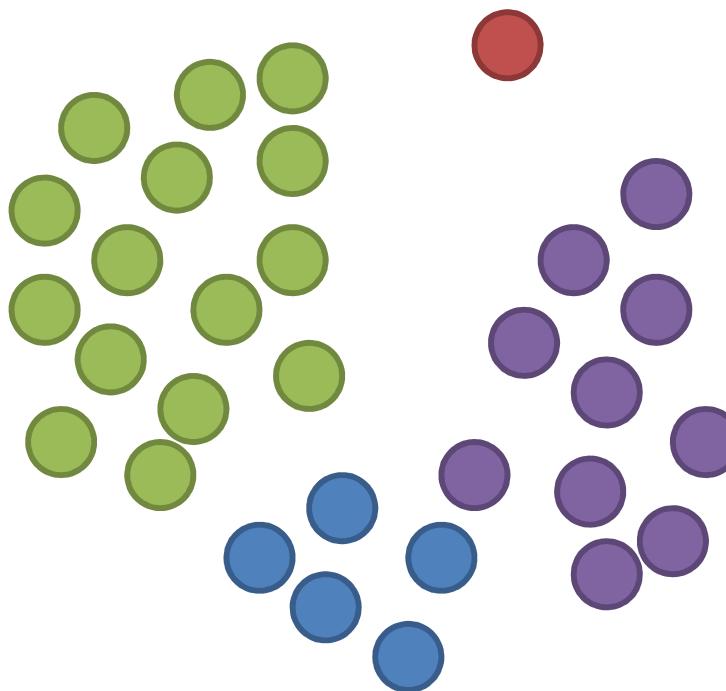
Grupowanie - DBSCAN



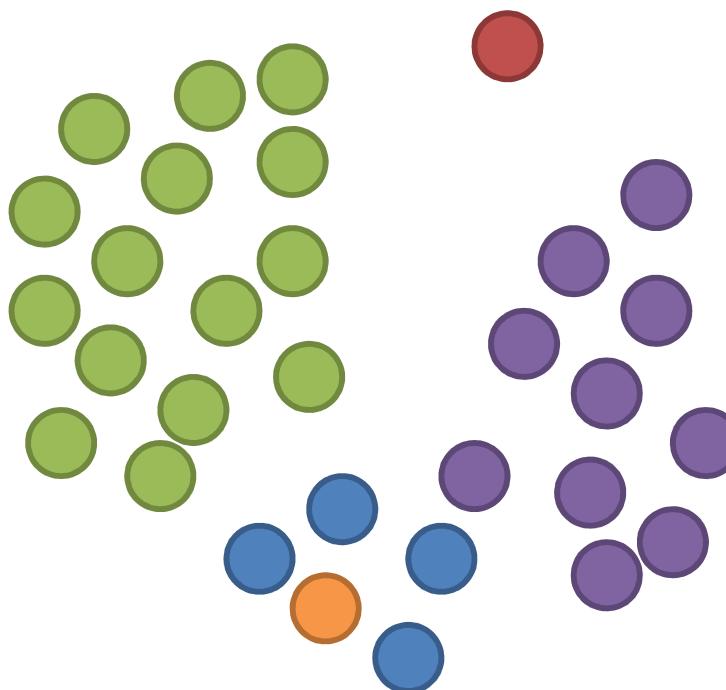
Grupowanie - DBSCAN



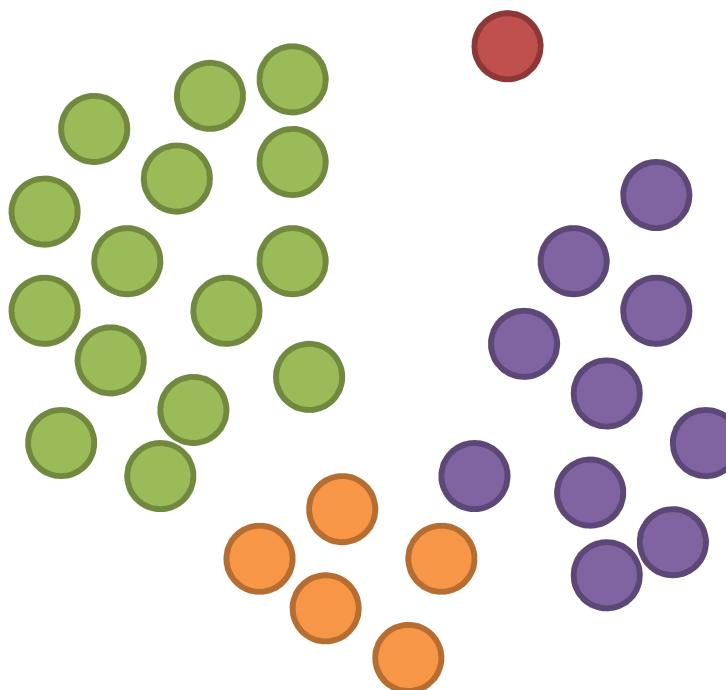
Grupowanie - DBSCAN



Grupowanie - DBSCAN



Grupowanie - DBSCAN



Teoria gier

Gry dzielimy ze względu na:

- **liczbę graczy:**

- 1-osobowe, bez przeciwników (np. pasjans, 15-tka, gra w życie, itp.),
- 2-osobowe (np. szachy, warcaby, go, itp.),
- wieloosobowe (np. brydż, giełda, itp.);

- **wygraną/przegraną:**

- o sumie zerowej (suma wygranych jest dokładnie równa sumie przegranych),
- o sumie niezerowej (np. dylemat więźnia, lotto, itp.);

- **współpracę:**

- kooperacyjne (np. gospodarka),
- niekooperacyjne (np. ucieczka-pościg, itp.);

- **rolę losowości:**

- całkiem losowe (np. lotto, ruletka, itp.),
- częściowo losowe (np. brydż, itp.),
- deterministyczne (np. szachy, itp.);

- **wiedzę graczy o stanie.**

Wykład 7, 31 III 2010, str. 2

Teoria gier

Odróżniać **losowość** od **wiedzy graczy o stanie!**

	pełna wiedza graczy o stanie	niepełna wiedza graczy o stanie
determinizm	szachy	okręty
losowość	gry z kostką (np. chińczyk)	brydż

Do opisu gry potrzeba:

- specyfikacji graczy,
- określenia ich celów,
- opisu dostępnej informacji,
- specyfikacji ich strategii.

Teoria gier

Prosta gra macierzowa, 2-osobowa, o sumie zerowej:

- gracze wykonują ruchy jednocześnie;
- macierz wypłat dla A (skoro suma zerowa, wystarczy podać, ile wygrywa jeden gracz, drugi przegrywa tyle samo):

		ruchy gracza B			
		b_1	b_2	\dots	b_k
ruchy gracza A	a_1	u_{11}	u_{12}	\dots	u_{1k}
	a_2	u_{21}	u_{22}	\dots	u_{2k}
	\vdots	\dots	\dots	\dots	\dots
	a_n	u_{n1}	u_{n2}	\dots	u_{nk}

$$u_{ij} \in \mathbb{R} \quad \text{dla} \\ \langle i, j \rangle \in [1..n] \times [1..k]$$

Teoria gier

Wykład 7, 31 III 2010, str. 4

Prosta gra macierzowa, 2-osobowa, o sumie zerowej:

PAPIER, KAMIENÍ, NOŽYCE

- gracze wykonują ruchy jednocześnie;
- macierz wypłat dla A :

A	B	papier	kamień	nożyce
papier	0	1	-1	
kamień	-1	0	1	
nożyce	1	-1	0	

Prosta gra macierzowa, 2-osobowa, o sumie niezerowej:

DYLEMAT WIĘŹNIA

- gracze wykonują ruchy jednocześnie;
- macierz wypłat:

	<i>A</i> \ <i>B</i>	współpraca	zdrada
współpraca		-3 \ -3	-20 \ 0
zdrada	0 \ -20	-10 \ -10	

Strategia optymalna: **zawsze zdradzać**.

Wykład 7, 31 III 2010, str. 6

Dygresja polityczno-ewolucyjna

Mantra liberalna:

Niech każdy dba o swój interes,
wtedy interes wspólny sam o siebie zadba.

W sytuacji dylematu więźnia (np. w sprawach globalnych, jak wpływ na klimat) interesy jednostkowe **nie składają się** w interes wspólny...

Problem ewolucjonistów:

Skąd w przyrodzie biorą się zachowania altruistyczne?

Geny osobnika, który poświęca się dla bliźniego/grupy/społeczności, powinny zanikać, bo to jest **gorsza strategia** gry niż egoizm...

Dygresja polityczno-ewolucyjna

Hipotetyczna odpowiedź:

Iterowany dylemat więźnia:

Gramy wiele razy i w każdej rozgrywce bierzemy pod uwagę zachowanie przeciwnika/partnera w poprzednich rozgrywkach.

Jeśli liczba rozgrywek nie jest z góry ograniczona, to strategią **lepszą od pełnego egoizmu** jest „**wet za wet**”:

- na początku współpracować,
 - w następnych turach robić tak, jak poprzednio zrobił przeciwnik;
 - co jakiś czas (losowo) „darować winę” — pójść na współpracę mimo że przeciwnik zdradził.

Być może liberalny ład społeczny może działać na zasadzie iterowanego dylematu więźnia... .

Być może altruizm powstaje z iterowanego dylematu więźnia...

Wykład 7, 31 III 2010, str. 8

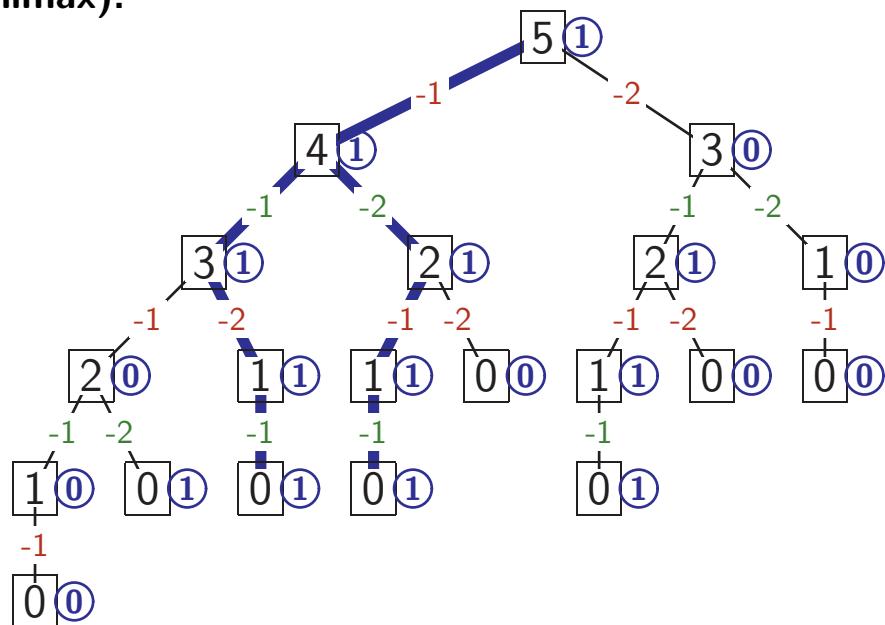
Teoria gier

Gra w zapałki (minimax):

Gracze kolejno biorą z kupki po 1 lub 2 zapałki.

Kto weźmie ostatnią
— przegrywa.

Wygrana



Czerwony zainteresowany jest maksymalizacją wyniku.

Zielony zainteresowany jest **minimalizacja wyniku**.

Teoria gier

Strategia „minimax”:

- **minimax** stosuje się do gier 2-osobowych, w których gracze wykonują ruchy na przemian; jeden z nich zainteresowany jest **minimalizacją** a drugi **maksymalizacją** wyniku;
- ocenia się wszystkie pozycje w pełnym **drzewie gry** — począwszy od liści i skończywszy na korzeniu:
 - najpierw na liściach wpisuje się wyniki zakończonych rozgrywek;
 - potem na każdym wierzchołku wewnętrznym wpisuje się **minimum** lub **maximum** (zależnie od tego, który gracz ma ruch) ocen dzieci tego wierzchołka;
- w każdej pozycji gracz powinien wykonać ruch prowadzący do pozycji o **najniższej** lub **najwyższej** (zależnie od tego, który z nich) ocenie.

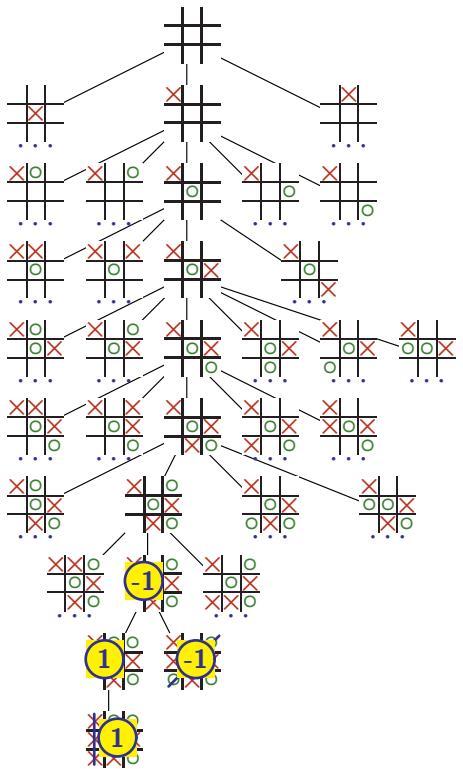
Wykład 7, 31 III 2010, str. 10

Teoria gier

Funkcja oceny pozycji w strategii „minimax”:

```
int ocena (Stan_planszy pl, int czyj_ruch) {  
    if (gra_zakonczona(pl, czyj_ruch))  
        return wielkosc_wyplaty(pl, czyj_ruch);  
    else {  
        int i, min, max, oc; Stan_planszy pl1;  
        min = infny; max = -infny;  
        for (i in zbior_mozliwych_ruchow(pl, czyj_ruch)) {  
            wykonaj_ruch(&pl1, pl, i);  
            oc = ocena(pl1, (czyj_ruch == MAX ? MIN : MAX));  
            if (oc > max) max = oc;  
            if (oc < min) min = oc;  
            if (czyj_ruch == MAX) return max;  
            else return min;  
        }  
    }  
}
```

Teoria gier



Wygrana

- krzyżyka: 1;
- kółka: -1.

Krzyżyk zainteresowany jest maksymalizacją.

Kółko zainteresowane jest minimalizacją.

Drzewa nawet prostych gier są na ogół olbrzymie, a minimax trzeba zacząć od liści, więc wydaje się, że potrzebne jest całe drzewo.

Teoria gier

Wykład 7, 31 III 2010, str. 12

Heurystyczna ocena pozycji:

- budujemy **fragment** drzewa gry o takiej wysokości, na jaką nas stać; liście tego fragmentu są wewnętrznymi węzłami całego drzewa, więc nie ma pewności, kto w nich wygrywa;
- stosujemy jakąś **heurystyczną** ocenę pozycji na liściach tego fragmentu;
- propagujemy na cały fragment ocenę z liści stosując zwykły **minimax**;
- wybieramy ruch wg minimaxu;
- usuwamy oceny; przy następnym ruchu budujemy nowy fragment drzewa sięgający głębiej i oceniamy jak wyżej.

Teoria gier

Gra w zapałki:

Heureza:

(liczba zapałek) mod 3

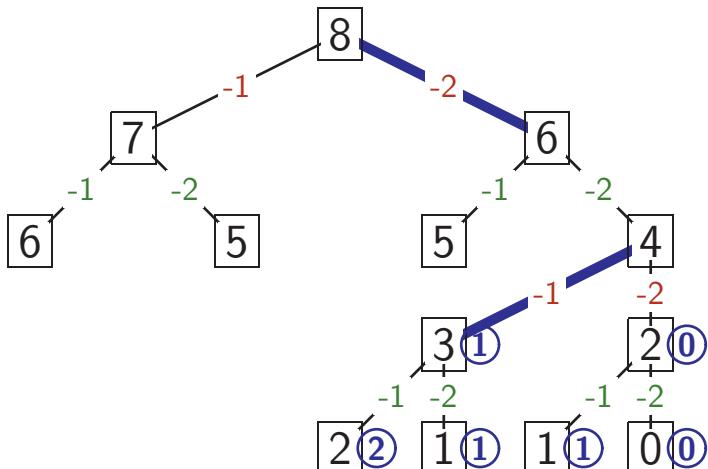
Czerwony: jak najwyższa ocena

Zielony: jak najniższa ocena

Uwaga:

Dla tej gry taka heureza nie ma sensu; to tylko taka sobie

ilustracja działania algorytmu.



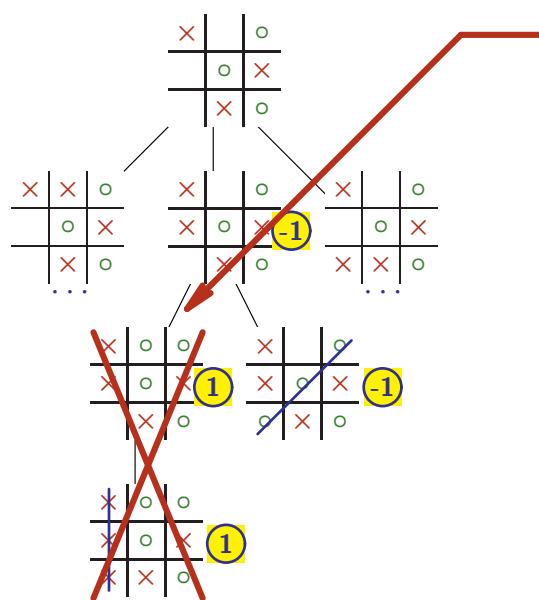
Z pktu widzenia Czerwonego:

1. biorę 2 zapałki; i sadzę, że Zielony weźmie 2;
2. biorę 1 zapałkę; i sadzę, że Zielony weźmie 2;
3. ...

Wykład 7, 31 III 2010, str. 14

Teoria gier

Przycinanie α - β :

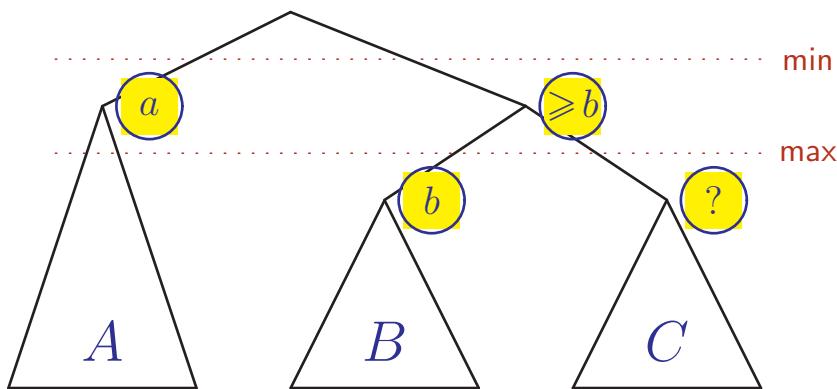


- ruch ma kółko, zainteresowany minimalizacją;
- to poddrzewo nie będzie grało, bo ma wyższą ocenę niż sąsiednie.

W takiej sytuacji tego poddrzewa można w ogóle nie rozpatrywać.

Teoria gier

Przycinanie α - β :



Jeśli $a \leq b$, to drzewa C nie trzeba przeglądać,
bo gracz **min** nie pozwoli tam wejść, wybierze poddrzewo A .

SZTUCZNA INTELIGENCJA I SYSTEMY DORADCZE

PRZESZUKIWANIE PRZESTRZENI STANÓW — GRY

Gry a problemy przeszukiwania

“Nieprzewidywalny” przeciwnik \Rightarrow rozwiązanie jest strategią specyfikującą posunięcie dla każdej możliwej odpowiedzi przeciwnika

Ograniczenia czasowe \Rightarrow Mało prawdopodobne znalezienie celu, trzeba aproksymować

Historia:

- Komputer rozważa różne scenariusze rozgrywki (Babbage, 1846)
- Algorytmy dla gier z pełną inform. (Zermelo, 1912; Von Neumann, 1944)
- Skończony horyzont, aproksymacyjna ocena stanu gry (Zuse, 1945; Wiener, 1948; Shannon, 1950)
- Pierwszy program grający w szachy (Turing, 1951)
- Zastosowanie uczenia maszynowego do poprawy trafności oceny stanu gry (Samuel, 1952–57)
- Odcięcia umożliwiające głębsze przeszukiwanie (McCarthy, 1956)

Rodzaje gier

	deterministyczne	niedeterministyczne
Pełna informacja	szachy, warcaby, go, otello	backgammon, monopoly
Niepełna informacja		bridge, poker, scrabble, nuclear war

Gra deterministyczna: 2 graczy

Gracze: **MAX** i **MIN**

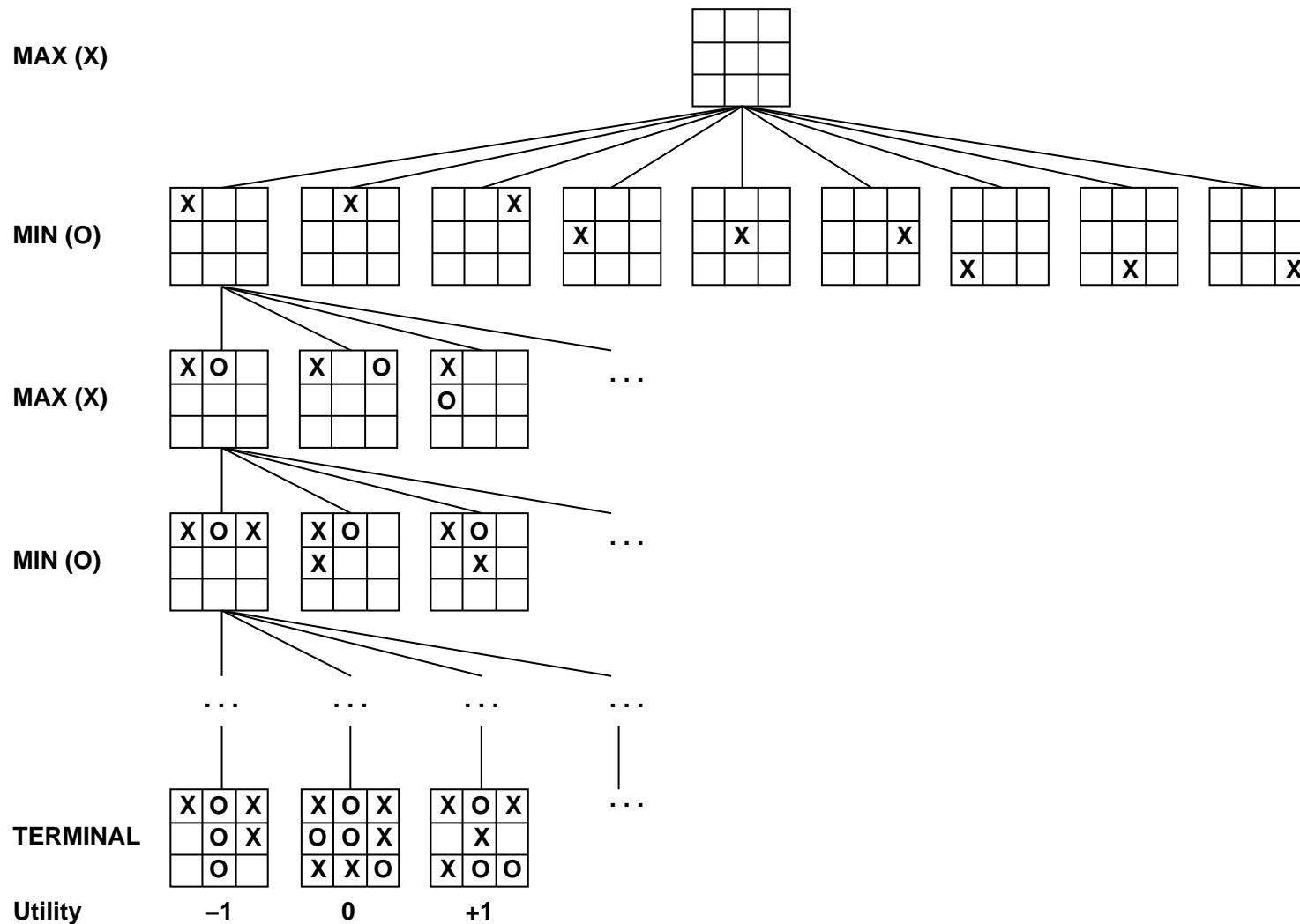
Stan początkowy: stan planszy i wskazanie gracza rozpoczynającego (**MAX**)

Funkcja następnika: zbiór par (*posunięcie, stan*) opisujących wszystkie dopuszczalne posunięcia z bieżącego stanu

Test końca gry: sprawdza, czy stan gry jest końcowy

Funkcja użyteczności (wypłaty): numeryczna wartość dla stanów końcowych np. wypłaty dla wygranej, porażki i remisu mogą być odpowiednio +1, -1 i 0.

Drzewo gry deterministycznej: 2 graczy



Strategia minimax: algorytm

Dla gier deterministycznych z pełną informacją

Pomysł: wybiera ruch zapewniający największą wypłatę

tzn. największą *wartość minimax* (funkcja MINIMAX-VALUE)
przy założeniu, że przeciwnik gra optymalnie

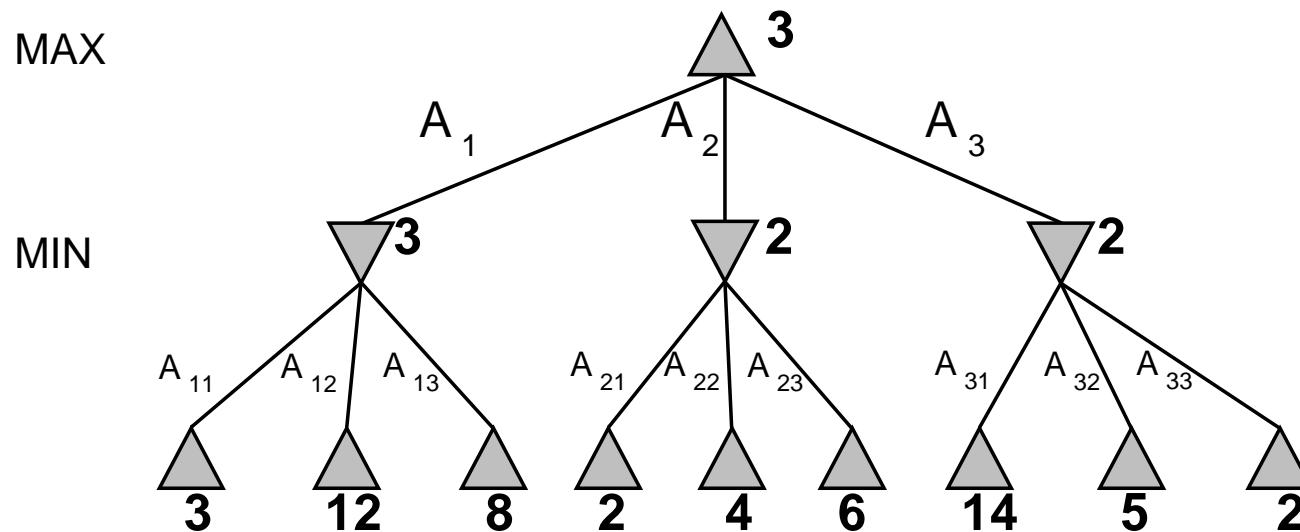
```
function MINIMAX-DECISION(state, game) returns an action
    action, state  $\leftarrow$  the a, s in SUCCESSORS(state)
        such that MINIMAX-VALUE(s, game) is maximized
    return action
```

```
function MINIMAX-VALUE(state, game) returns a utility value
    if TERMINAL-TEST(state) then
        return UTILITY(state)
    else if MAX is to move in state then
        return the highest MINIMAX-VALUE of SUCCESSORS(state)
    else
        return the lowest MINIMAX-VALUE of SUCCESSORS(state)
```

Strategia minimax: przykład

Gracz **MAX** maksymalizuje funkcję wypłaty (węzły \triangle)
⇒ wybiera ruch w lewą gałąź drzewa

Gracz **MIN** minimalizuje funkcję wypłaty (węzły ∇)
⇒ wybiera ruch do lewego liścia poddrzewa



Strategia minimax: własności

Użyteczność??

Strategia minimax: własności

Użyteczność?? Gry determinist. z pełną informacją z dowolną liczbą graczy

Pełność??

Strategia minimax: własności

Użyteczność?? Gry determinist. z pełną informacją z dowolną liczbą graczy

Pełność?? Tak, jeśli drzewo przeszukiwań jest skończone

Gry z nieskończonym drzewem przesz. mogą mieć strategie skończone!

Optymalność??

Strategia minimax: własności

Użyteczność?? Gry determinist. z pełną informacją z dowolną liczbą graczy

Pełność?? Tak, jeśli drzewo przeszukiwań jest skończone

Gry z nieskończonym drzewem przesz. mogą mieć strategie skończone!

Optymalność?? Tak, jeśli przeciwnik gra optymalnie

W ogólności nieoptymalne

Złożoność czasowa??

Strategia minimax: własności

Użyteczność?? Gry determinist. z pełną informacją z dowolną liczbą graczy

Pełność?? Tak, jeśli drzewo przeszukiwań jest skończone

Gry z nieskończonym drzewem przesz. mogą mieć strategie skończone!

Optymalność?? Tak, jeśli przeciwnik gra optymalnie

W ogólności nieoptymalne

Złożoność czasowa?? $O(b^m)$

Złożoność pamięciowa??

Strategia minimax: własności

Użyteczność?? Gry determinist. z pełną informacją z dowolną liczbą graczy

Pełność?? Tak, jeśli drzewo przeszukiwań jest skończone

Gry z nieskończonym drzewem przesz. mogą mieć strategie skończone!

Optymalność?? Tak, jeśli przeciwnik gra optymalnie

W ogólności nieoptymalne

Złożoność czasowa?? $O(b^m)$

Złożoność pamięciowa?? $O(bm)$ (przezszukiwanie wgłąb)

Strategia minimax: własności

Użyteczność?? Gry determinist. z pełną informacją z dowolną liczbą graczy

Pełność?? Tak, jeśli drzewo przeszukiwań jest skończone

Gry z nieskończonym drzewem przesz. mogą mieć strategie skończone!

Optymalność?? Tak, jeśli przeciwnik gra optymalnie

W ogólności nieoptymalne

Złożoność czasowa?? $O(b^m)$

Złożoność pamięciowa?? $O(bm)$ (przezszukiwanie wgłąb)

Dla szachów, $b \approx 35$, $m \approx 100$ dla "sensownych" rozgrywek

\Rightarrow dokładne rozwiązanie zupełnie nieosiągalne

Strategia minimax z odcieciem

Problem:

brak czasu na pełne przeszukanie przestrzeni stanów
np. 100 sekund na posunięcie, szybkość 10^4 węzłów/sek
 $\Rightarrow 10^6$ węzłów na ruch

Rozwiązanie:

przeszukiwanie z odcięciem ograniczającym głębokość przeszukiwania

Strategia minimax z odcieciem: algorytm

```
function MINIMAX-DECISION(state, game) returns an action
    action, state  $\leftarrow$  the a, s in SUCCESSORS(state)
        such that MINIMAX-CUTOFF(s, game) is maximized
    return action

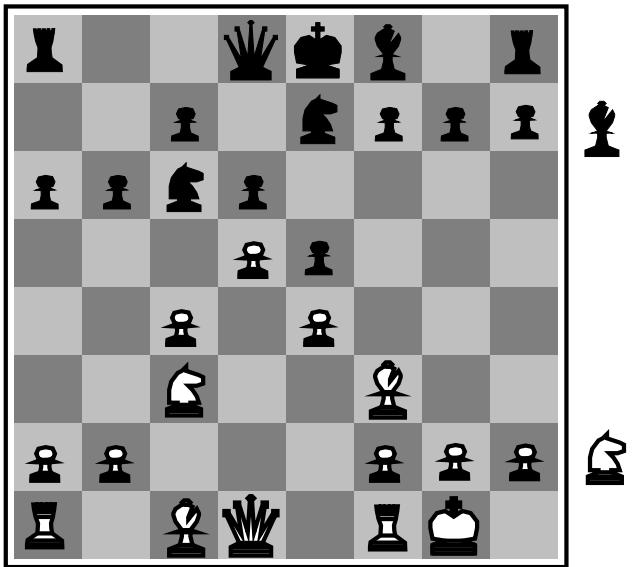


---


function MINIMAX-CUTOFF(state, game) returns a utility value
    if CUTOFF-TEST(state) then
        return EVAL(state)
    else if MAX is to move in state then
        return the highest MINIMAX-VALUE of SUCCESSORS(state)
    else
        return the lowest MINIMAX-VALUE of SUCCESSORS(state)
```

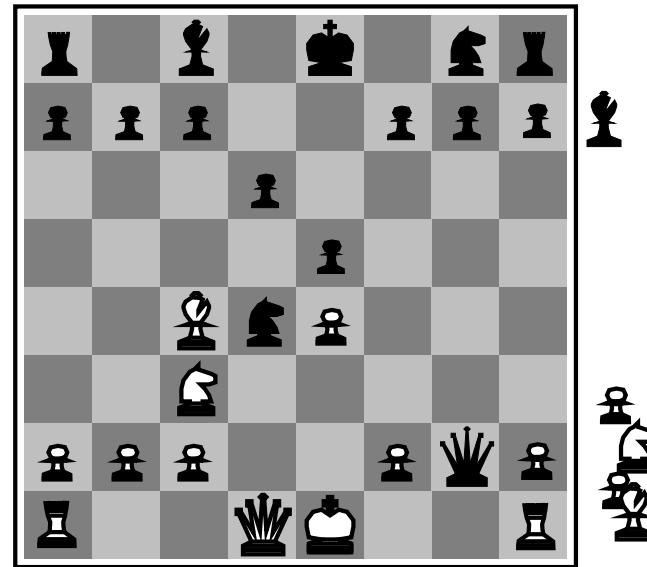
Funkcja oceny EVAL szacuje wypłatę dla danego stanu gry
= rzeczywistej wypłacie dla stanów końcowych

Funkcja oceny: przykład



Black to move

White slightly better



White to move

Black winning

Dla szachów, przeważnie *liniowa* ważona suma cech

$$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \dots + w_n f_n(s)$$

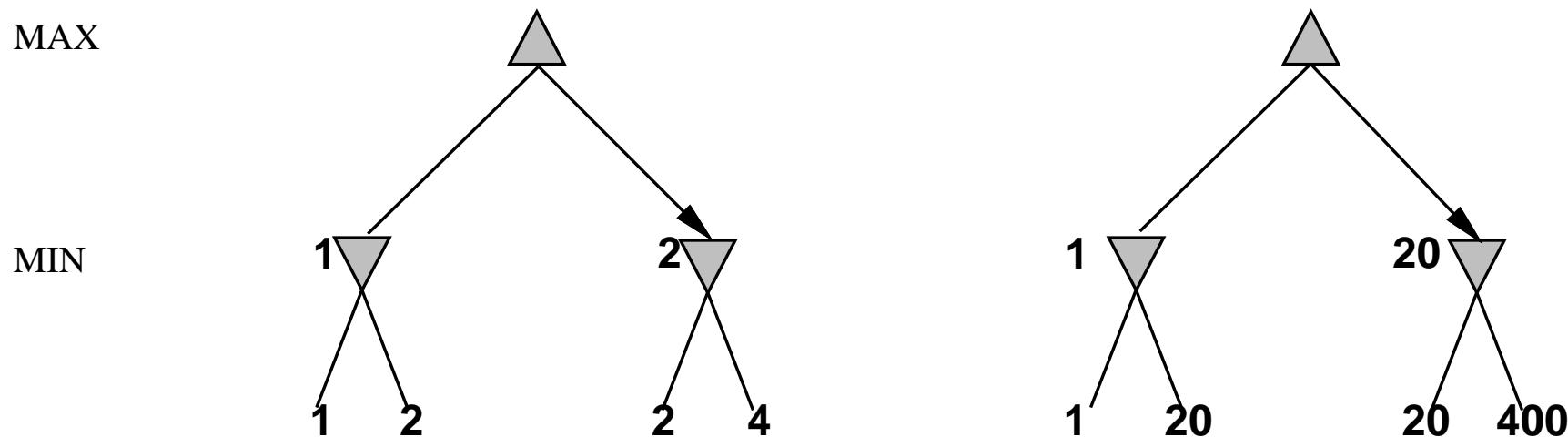
np. $w_1 = 9$ z

$f_1(s) = (\text{liczba białych hetmanów}) - (\text{liczba czarnych hetmanów})$, itd.

Strategia minimax z odcieciem: własności

Funkcja oceny wypłaty EVAL w grach deterministycznych ma znaczenie wyłącznie *porządkujące*

⇒ zachowuje działanie przy dowolnym przekształceniu *monotonicznym* funkcji EVAL



Strategia minimax z odcieciem: skuteczność

W praktyce dla szachów

$$b^m = 10^6, \quad b = 35 \quad \Rightarrow \quad m = 4$$

4-warstwowe przeszukiwanie \approx nowicjusz

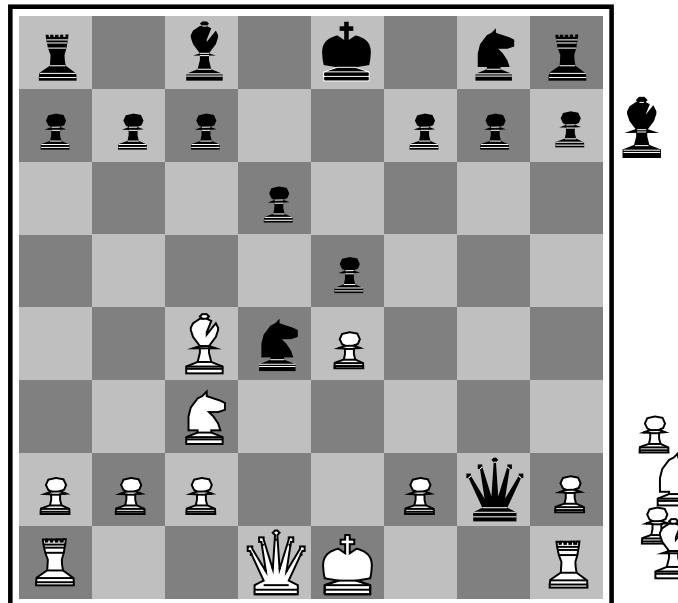
Potrzeba lepiej:

8-warstwowe przeszukiwanie \approx typowy PC, mistrz

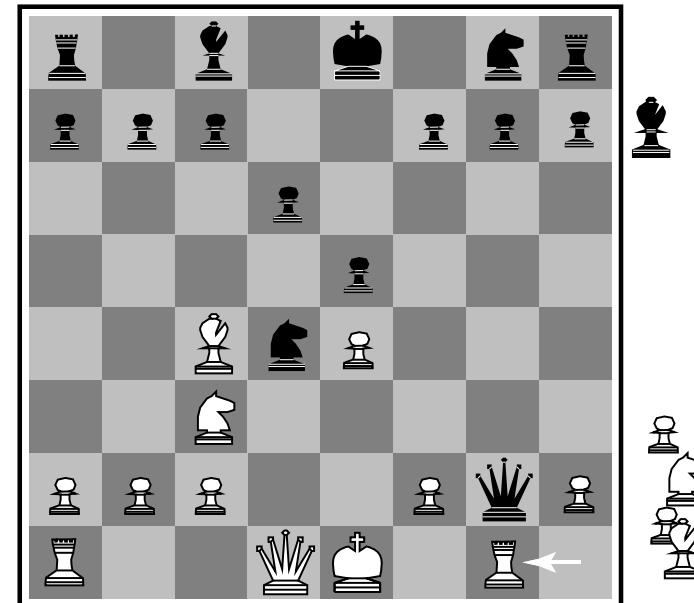
12-warstwowe przeszukiwanie \approx Deep Blue, Kasparov

Przeszukiwanie stabilne

Problem: Stany mają taką samą wartość oceny (na korzyść czarnych), ale stan z prawej *niestabilny*: kolejny ruch daje dużą zmianę oceny stanu gry (na korzyść białych)



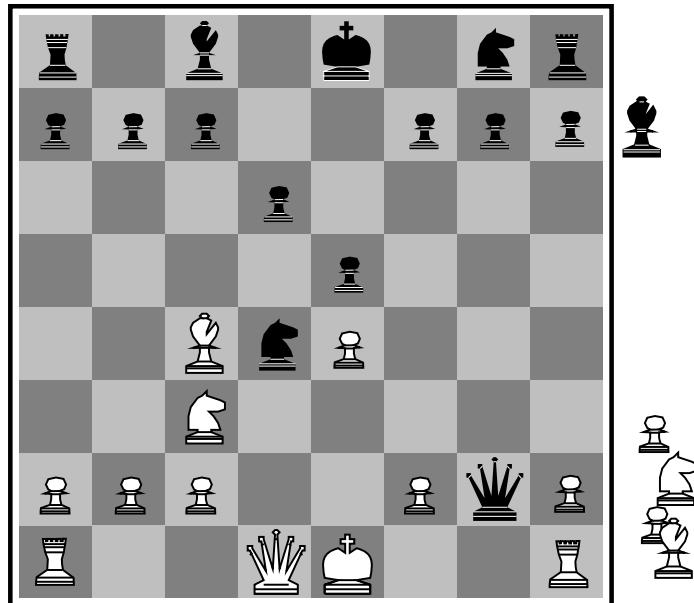
(a) White to move



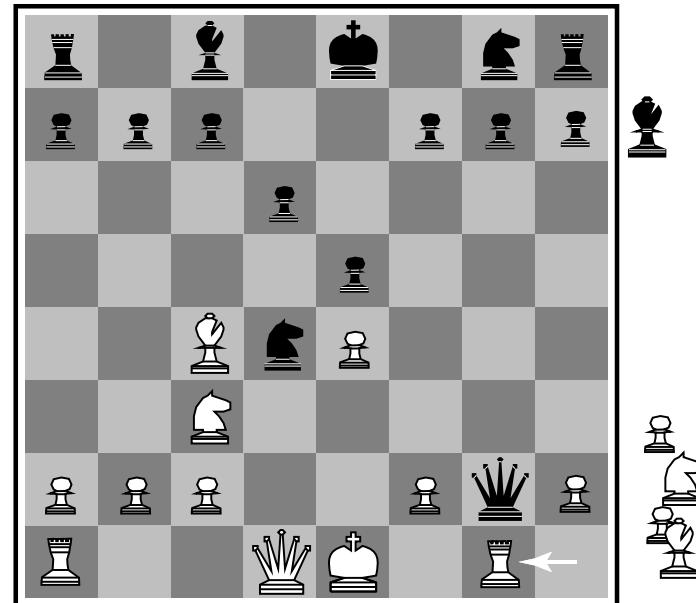
(b) White to move

Przeszukiwanie stabilne

Problem: Stany mają taką samą wartość oceny (na korzyść czarnych), ale stan z prawej *niestabilny*: kolejny ruch daje dużą zmianę oceny stanu gry (na korzyść białych)



(a) White to move

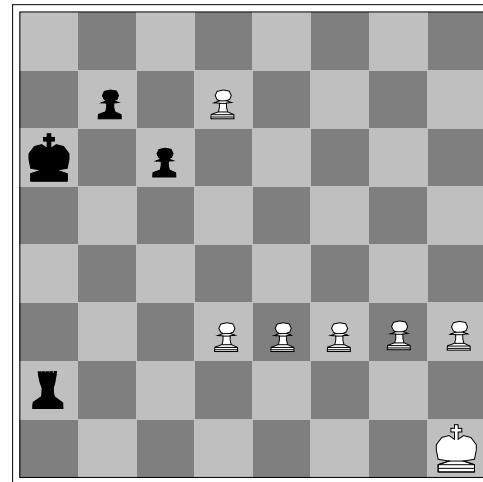


(b) White to move

Rozwiązanie: przeszukiwanie *stabilne*
stany niestabilne są rozwijane do momentu osiągnięcia stanu stabilnego

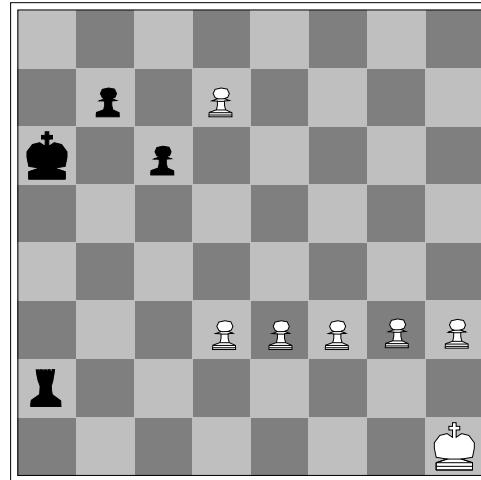
Przeszukiwanie z pojedynczym rozwinieciem

Efekt horyzontu: gracz wykonuje ruchy odsuwając nieuniknione posunięcie na korzyść przeciwnika poza *horyzont* przeszukiwania
np. czarna wieża powtarza szachowanie białego króla



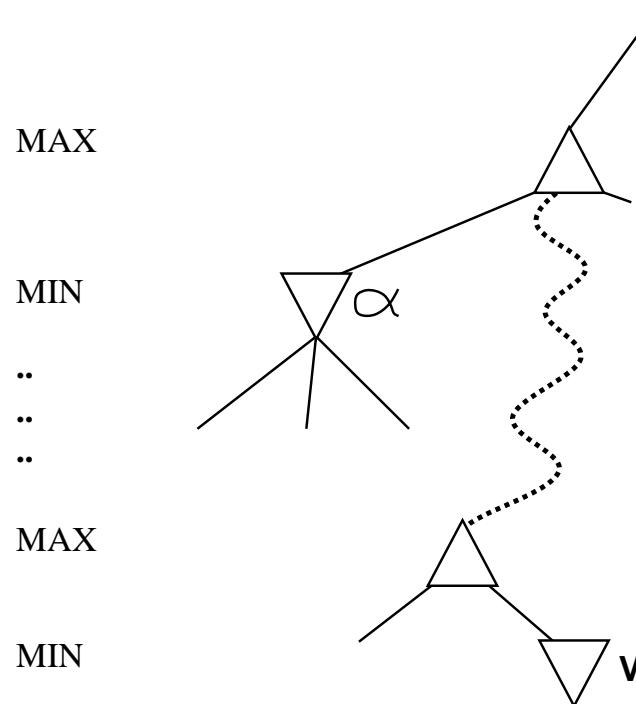
Przeszukiwanie z pojedynczym rozwinięciem

Efekt horyzontu: gracz wykonuje ruchy odsuwając nieuniknione posunięcie na korzyść przeciwnika poza *horyzont* przeszukiwania
np. czarna wieża powtarza szachowanie białego króla



Rozwiązanie: przeszukiwanie *z pojedynczym rozwinięciem*
algorytm wykonuje pogłębione przeszukiwanie
dla wybranych posunięć “wyraźnie lepszych” od pozostałych

Strategia minimax z odcięciem $\alpha-\beta$



α jest najlepszą wartością dla MAX poza bieżącą ścieżką przeszukiwania

Jeśli V jest gorsze niż α , MAX nigdy nie wejdzie do tej gałęzi
⇒ gałąź z V można odciąć

β jest definiowane analogicznie dla MIN

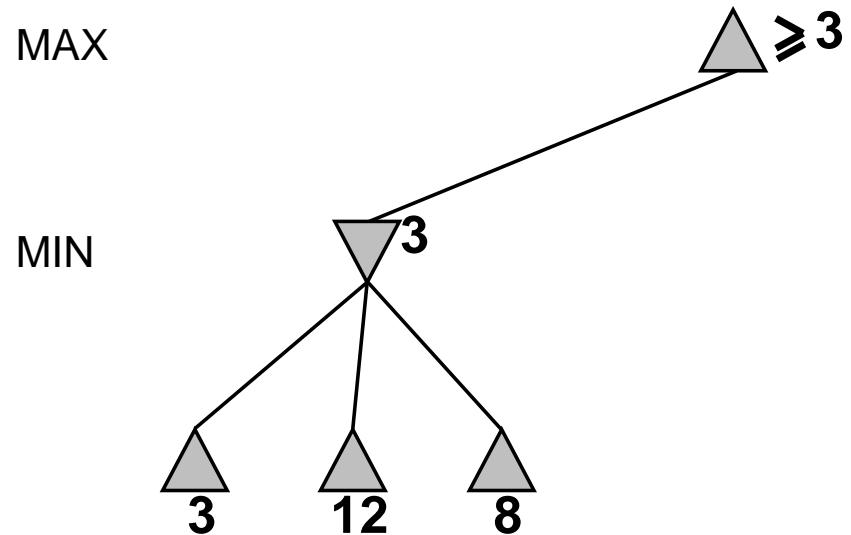
Strategia minimax z odcieciem α - β : algorytm

```
function ALPHA-BETA-SEARCH(state, game) returns an action
    action, state  $\leftarrow$  the a, s in SUCCESSORS[game](state)
        such that MIN-VALUE(s, game, -∞, +∞) is maximized
    return action
```

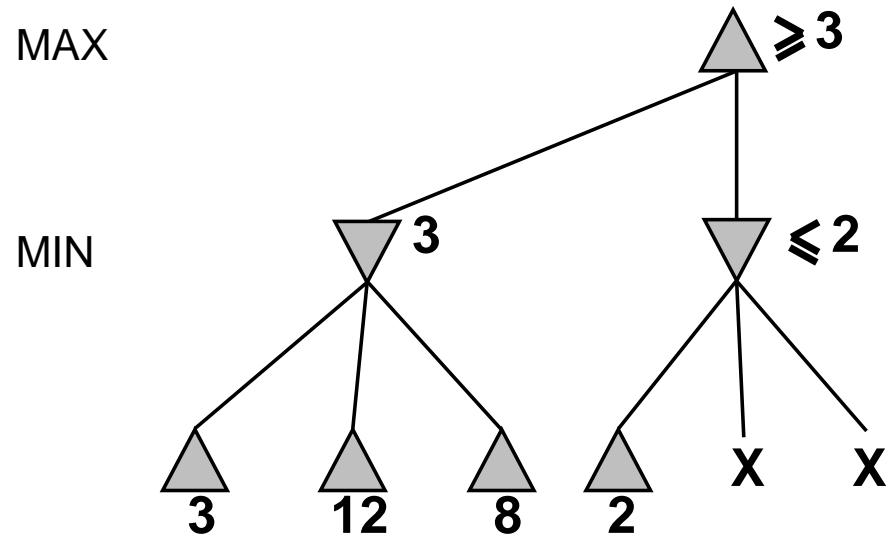
```
function MAX-VALUE(state, game, α, β) returns the minimax value of state
    if CUTOFF-TEST(state) then return EVAL(state)
    for each s in SUCCESSORS(state) do
         $\alpha \leftarrow \max(\alpha, \text{MIN-VALUE}(s, \text{game}, \alpha, \beta))$ 
        if  $\alpha \geq \beta$  then return  $\beta$ 
    return  $\alpha$ 
```

```
function MIN-VALUE(state, game, α, β) returns the minimax value of state
    if CUTOFF-TEST(state) then return EVAL(state)
    for each s in SUCCESSORS(state) do
         $\beta \leftarrow \min(\beta, \text{MAX-VALUE}(s, \text{game}, \alpha, \beta))$ 
        if  $\beta \leq \alpha$  then return  $\alpha$ 
    return  $\beta$ 
```

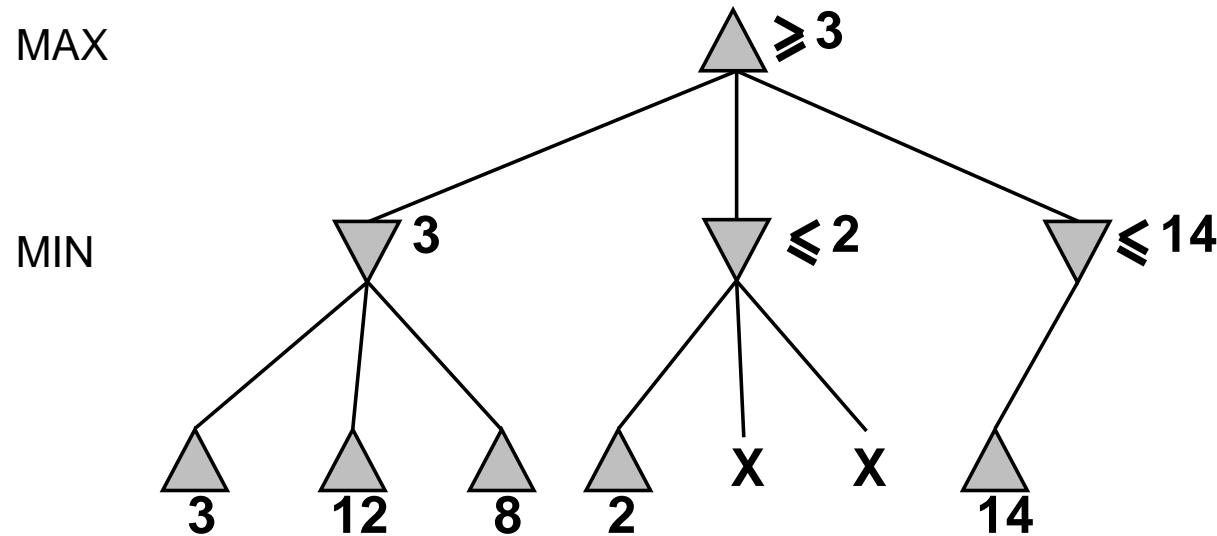
Strategia minimax z odcieciem $\alpha-\beta$: przykład



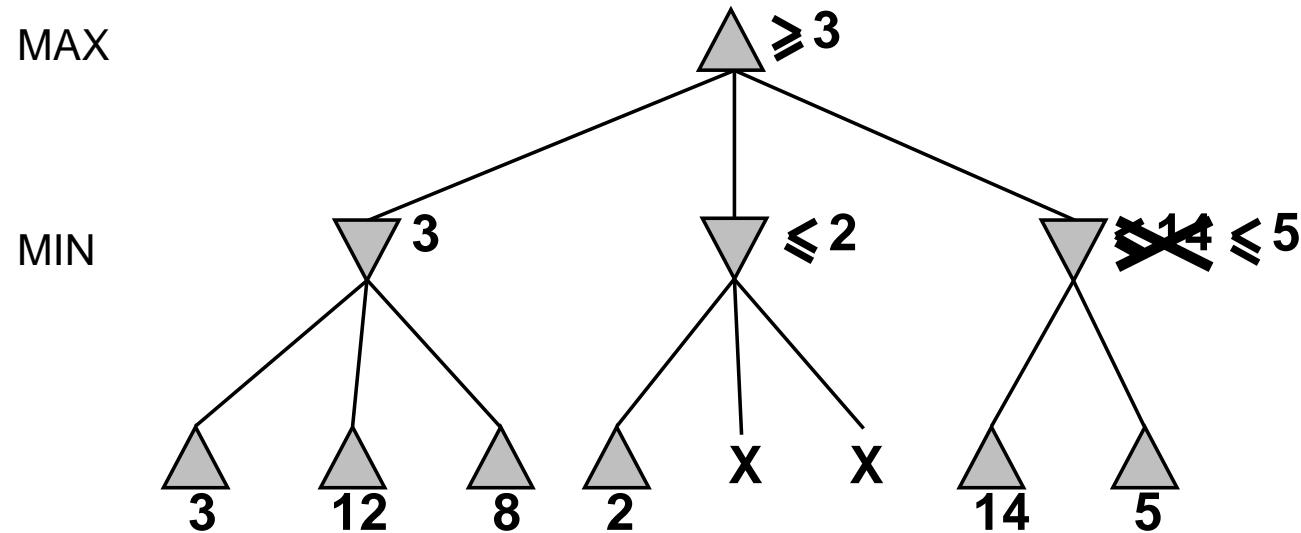
Strategia minimax z odcieciem $\alpha-\beta$: przykład



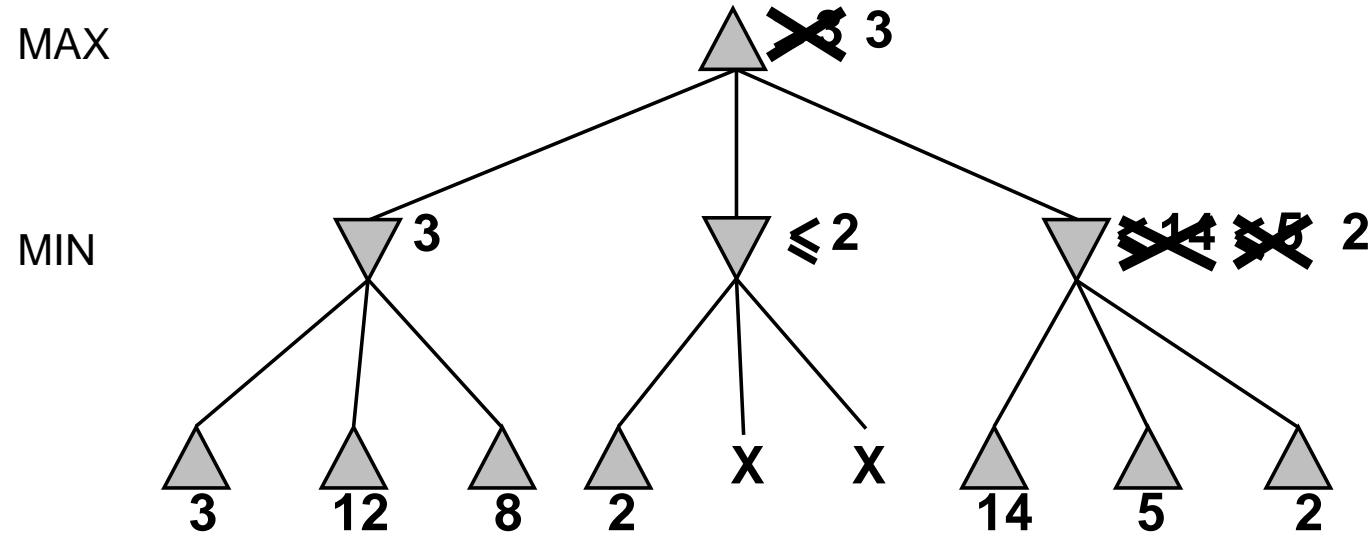
Strategia minimax z odcieciem $\alpha-\beta$: przykład



Strategia minimax z odcieciem $\alpha-\beta$: przykład



Strategia minimax z odcieciem $\alpha-\beta$: przykład



Strategia minimax z odcieciem $\alpha-\beta$: własności

Odcinanie jest "czyste": nie ma wpływu na optymalność i wynik przeszukiwania

Właściwe uporządkowanie posunięć poprawia efektywność odcinania

Dla "perfekcyjnego uporządkowania" posunięć złożoność czasowa = $O(b^{m/2})$

⇒ *podwaja* głębokość przeszukiwania

⇒ może łatwo zejść do 8-ego poziomu i grać dobre szachy

Gry deterministyczne: osiągnięcia

Warcaby: Chinook zakończył 40-letnie panowanie mistrza świata Mariona Tinsley w 1994. Użył biblioteki wszystkich zakończeń dla 8 lub mniej pionków na planszy, w sumie 443,748,401,247 pozycji.

Szachy: Deep Blue pokonał mistrza świata Gary Kasparowa w meczu z 6-ioma partiami w 1997. Deep Blue przeszukiwał 200 milionów pozycji na sekundę, używając bardzo wyszukanej funkcji oceny, i nieznanych metod rozszerzających niektóre ścieżki przeszukiwania do głębokości 40.

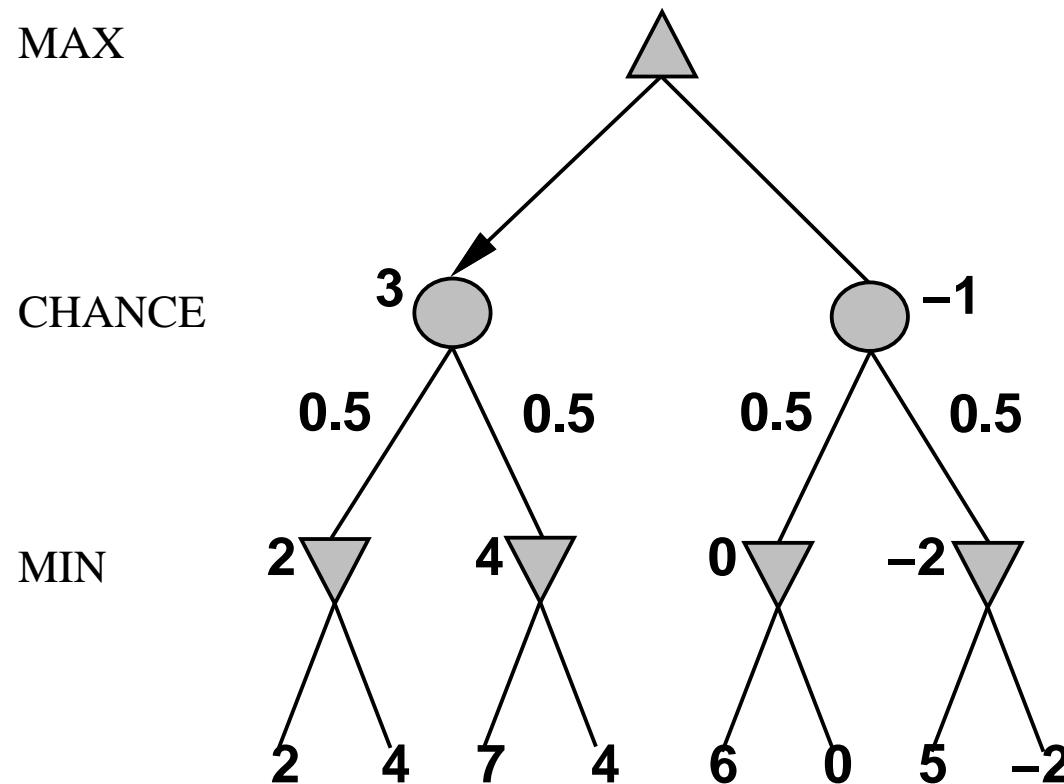
Otello: mistrz świata odmówił rozgrywki z komputerami, które są zbyt silne.

Go: mistrz świata odmówił rozgrywki z komputerami, które są zbyt słabe. W go, $b > 300$, więc większość programów używa bazy wiedzy z wzorcami do wyboru dopuszczalnych ruchów.

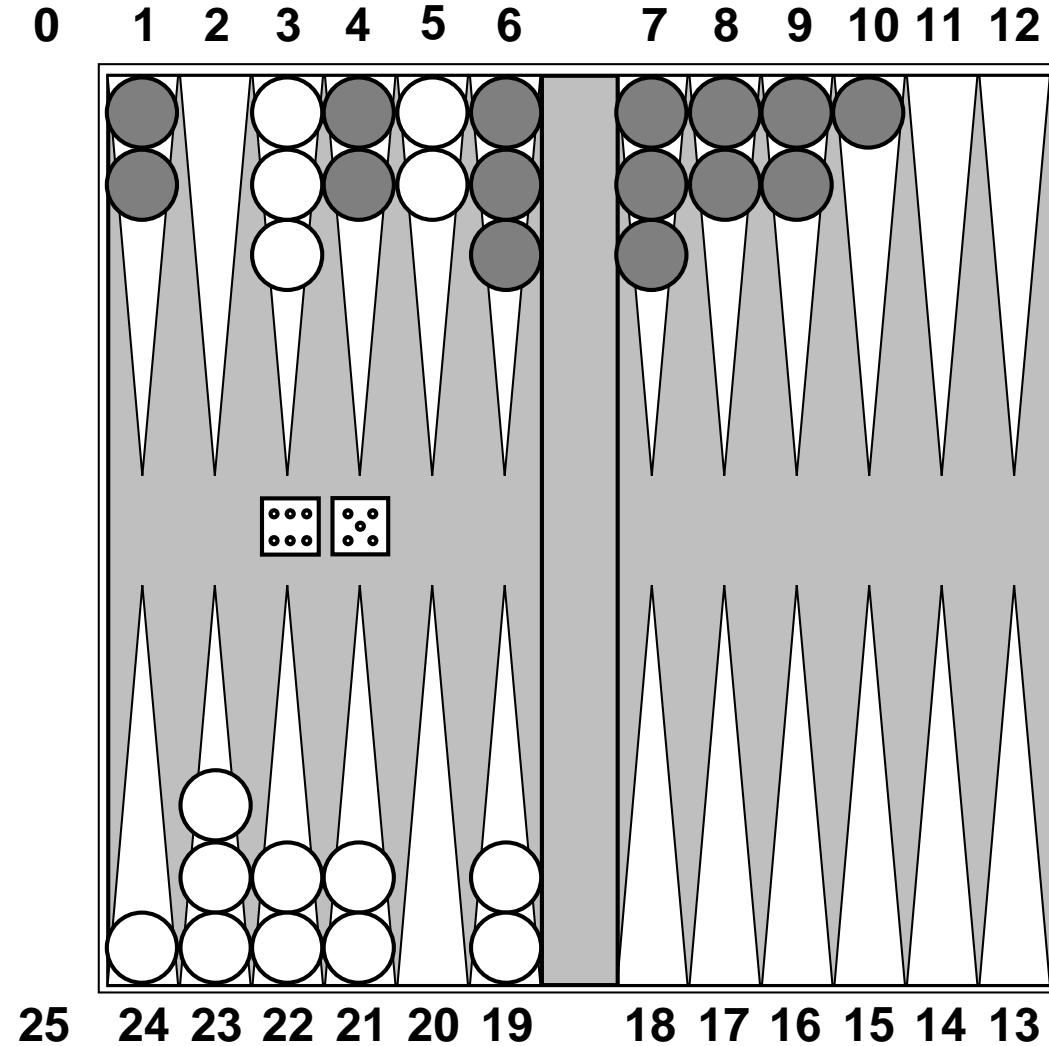
Gry niedeterministyczne

Źródło niedeterminizmu: rzut kostką, tasowanie kart

Przykład z rzucaniem monetą:



Gry niedeterministyczne: backgammon



Strategia usrednionego minimax

Uogólnienie strategii minimax dla gier niedeterministycznych

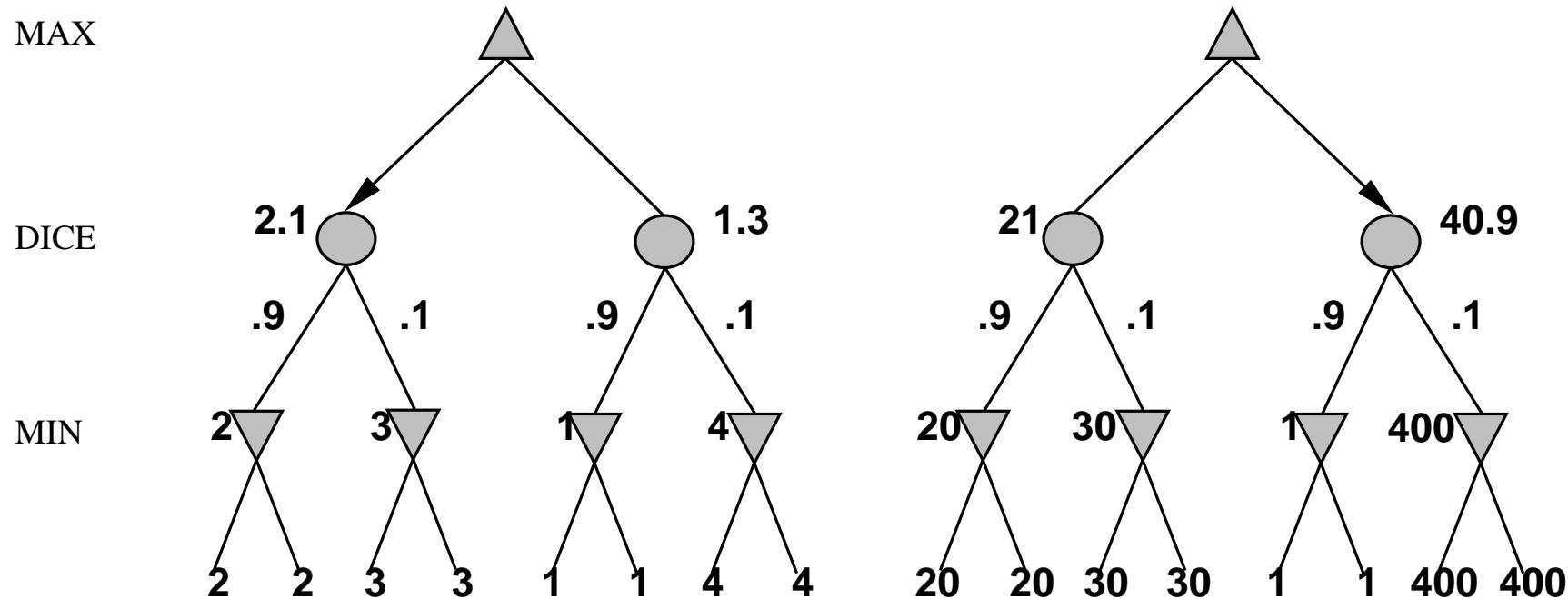
```
function EXPECTIMINIMAX-DECISION(state, game) returns an action
    action, state  $\leftarrow$  the a, s in SUCCESSORS(state)
        such that EXPECTIMINIMAX-VALUE(s, game) is maximized
    return action



---


function EXPECTIMINIMAX-VALUE(state, game) returns a utility value
    if TERMINAL-TEST(state) then
        return UTILITY(state)
    else if state is a MAX node then
        return the highest EXPECTIMINIMAX-VALUE of SUCCESSORS(state)
    else if state is a MIN node then
        return the lowest EXPECTIMINIMAX-VALUE of SUCCESSORS(state)
    else if state is a chance node then
        return average of EXPECTIMINIMAX-VALUE of SUCCESSORS(state)
```

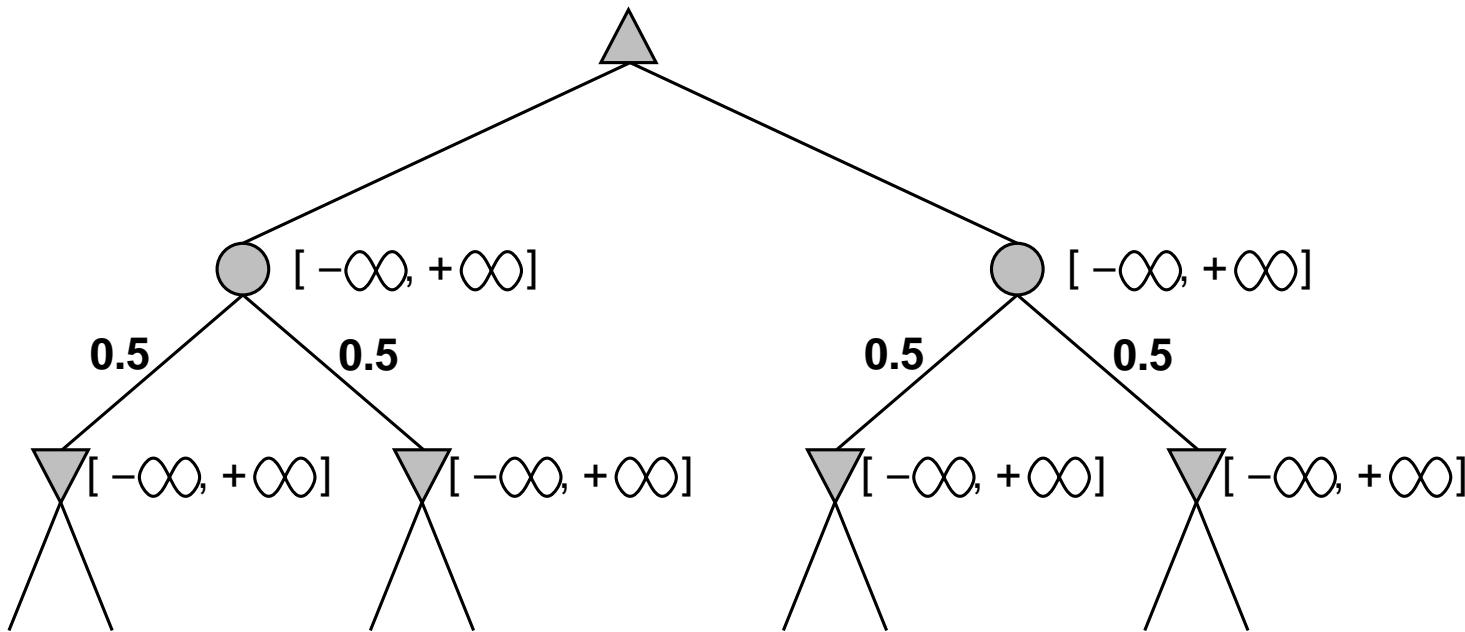
Strategia usrednionego minimax: własności



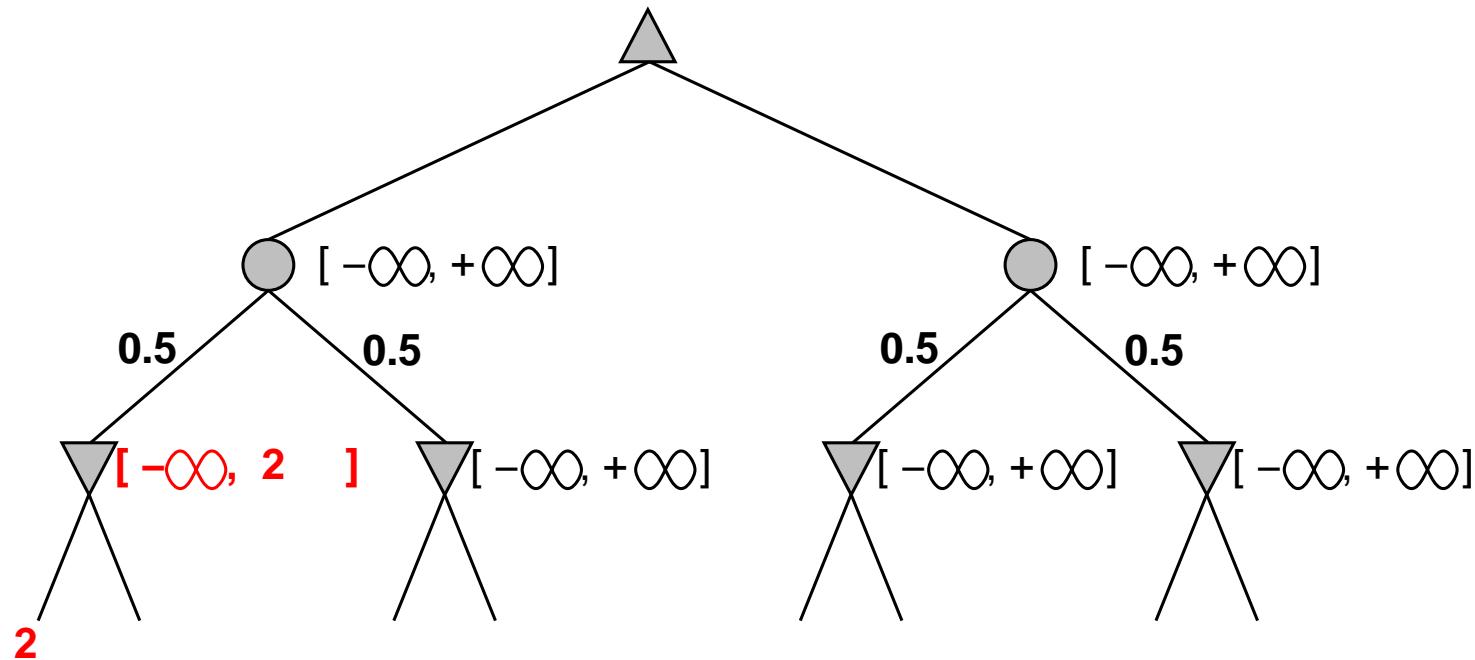
Działanie funkcji oceny EVAL jest zachowane tylko dla *dodatnich liniowych* przekształceń tej funkcji

Stąd EVAL powinna być proporcjonalna do wartości oczekiwanej wypłaty

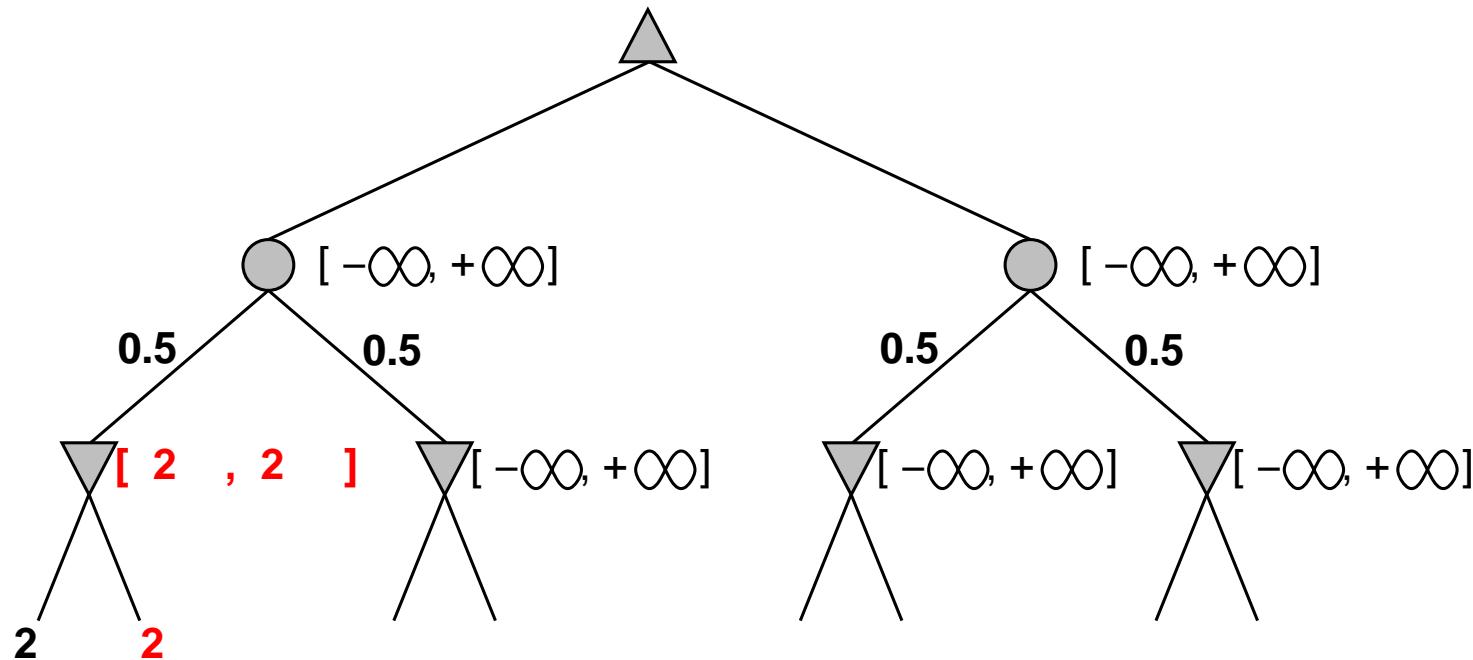
Strategia usrednionego minimax z odcieciem α - β



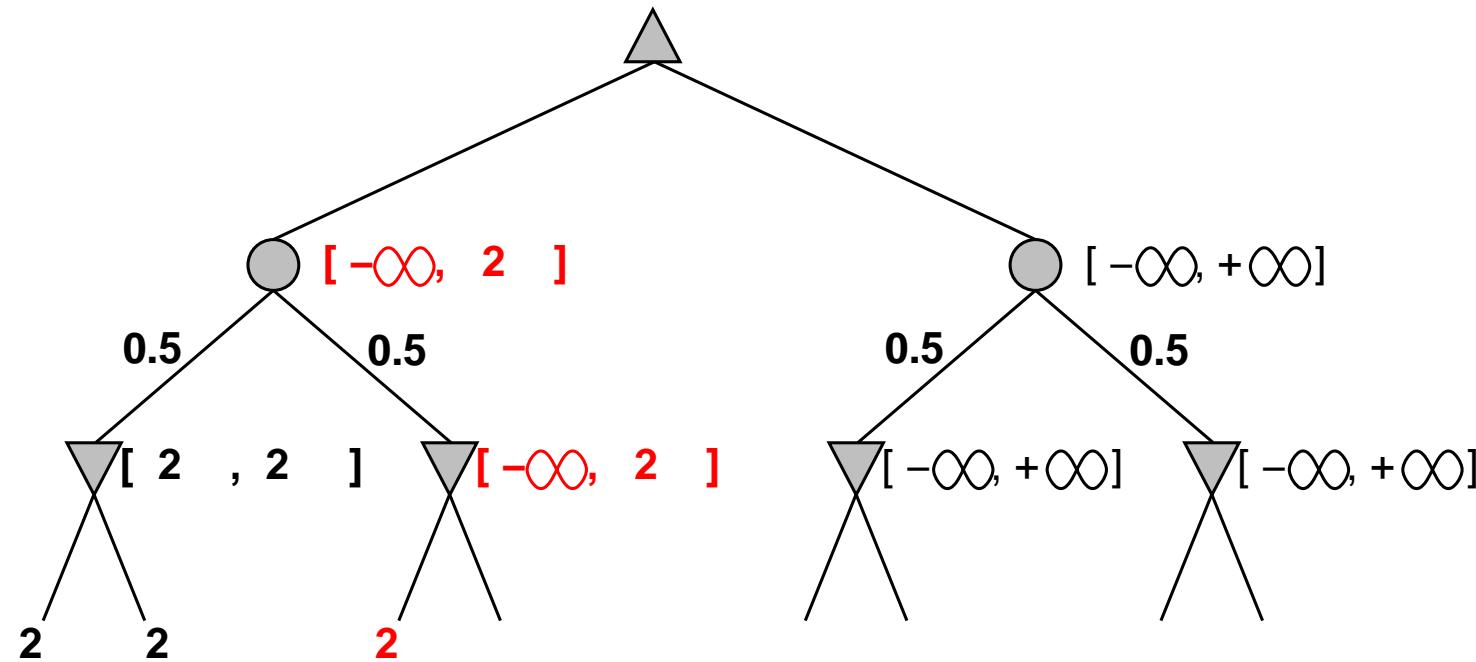
Strategia usrednionego minimax z odcieciem α - β



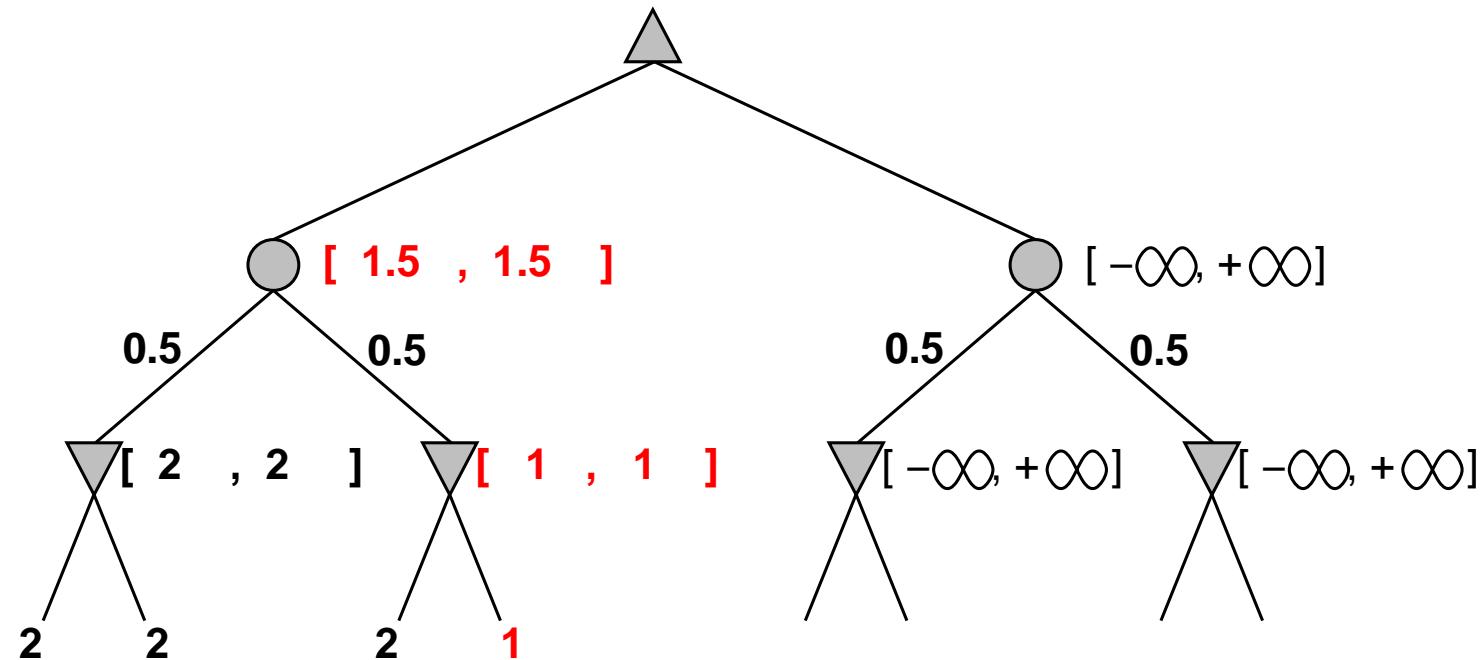
Strategia usrednionego minimax z odcieciem α - β



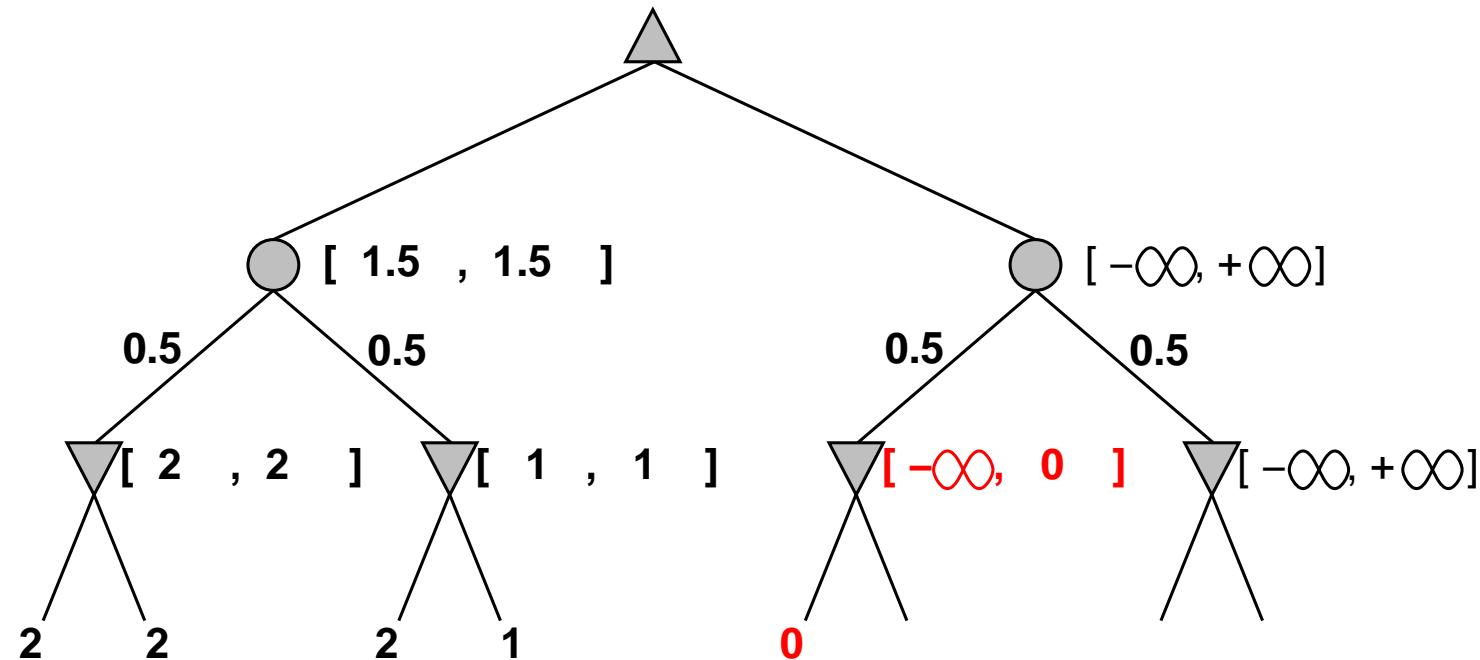
Strategia usrednionego minimax z odcieciem α - β



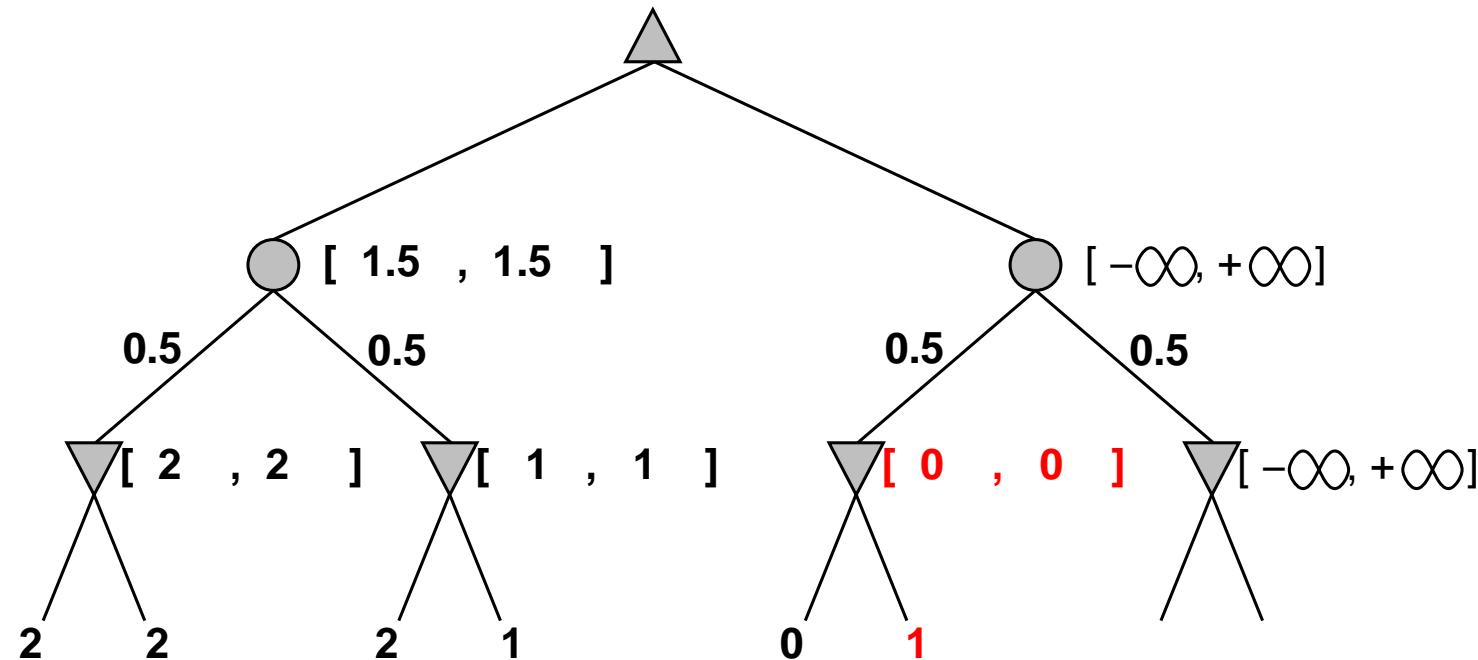
Strategia usrednionego minimax z odcieciem α - β



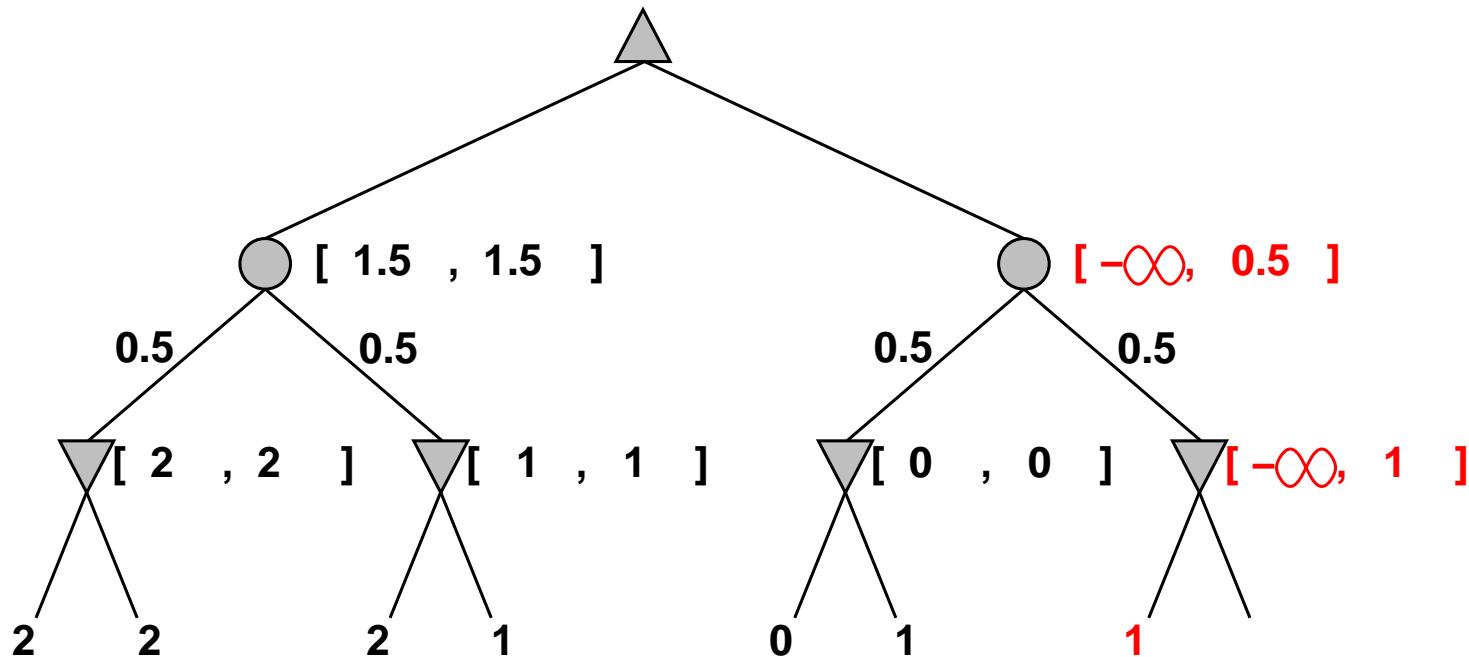
Strategia usrednionego minimax z odcieciem α - β



Strategia usrednionego minimax z odcieciem α - β

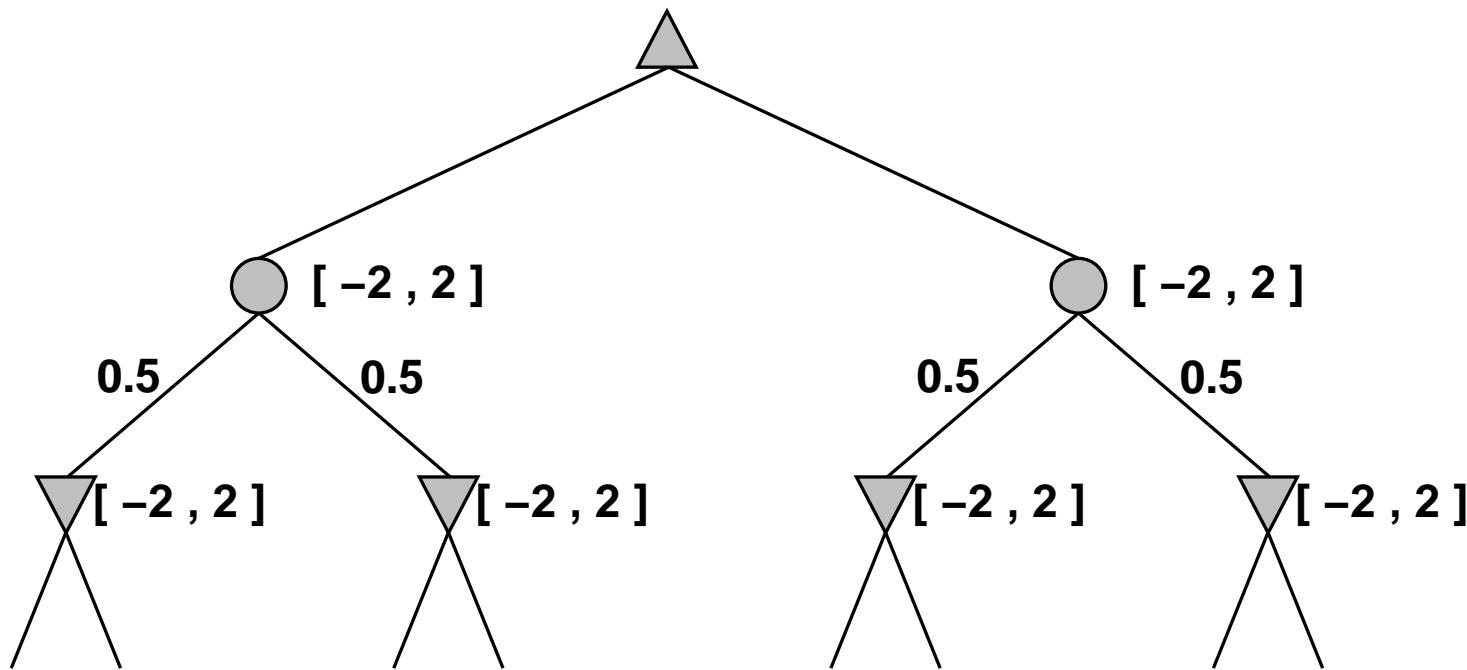


Strategia usrednionego minimax z odcieciem α - β



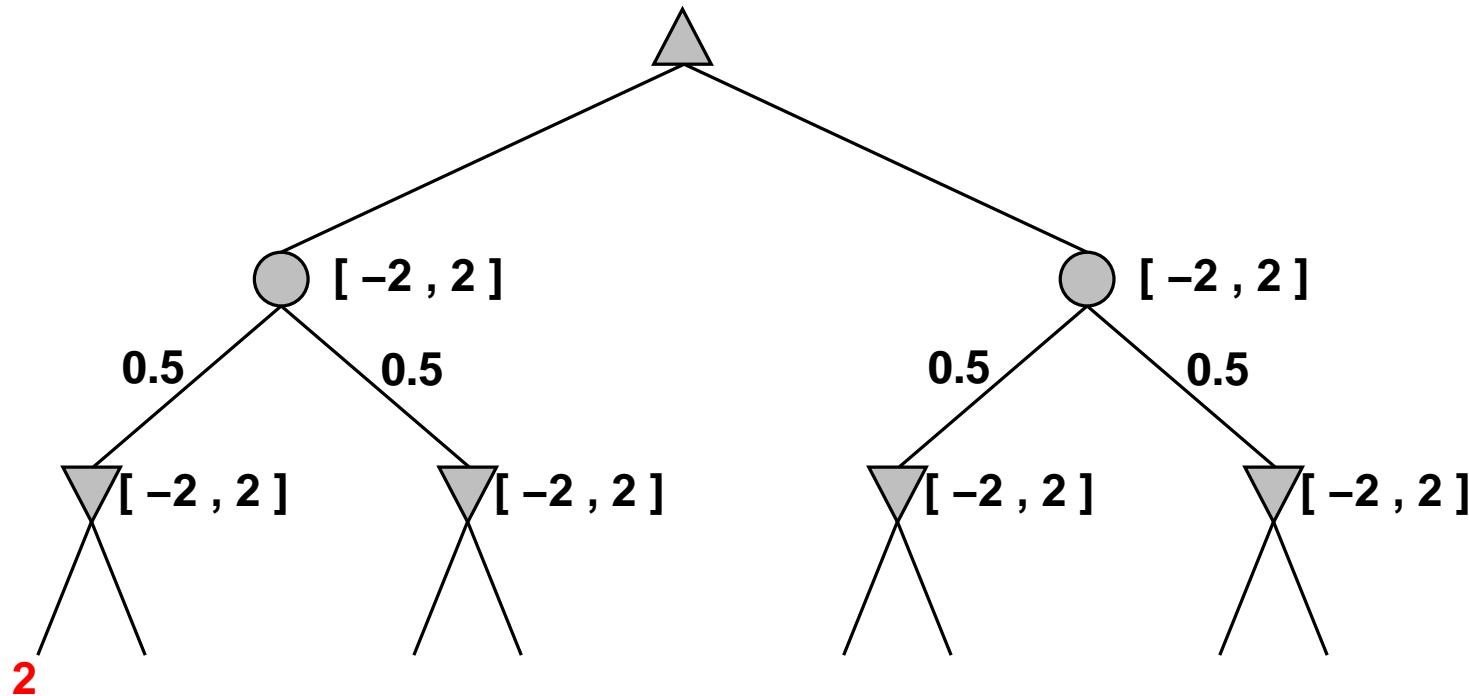
Strategia usrednionego minimax z odcieciem α - β

Bardziej efektywna, jeśli wartość wypłaty ograniczona



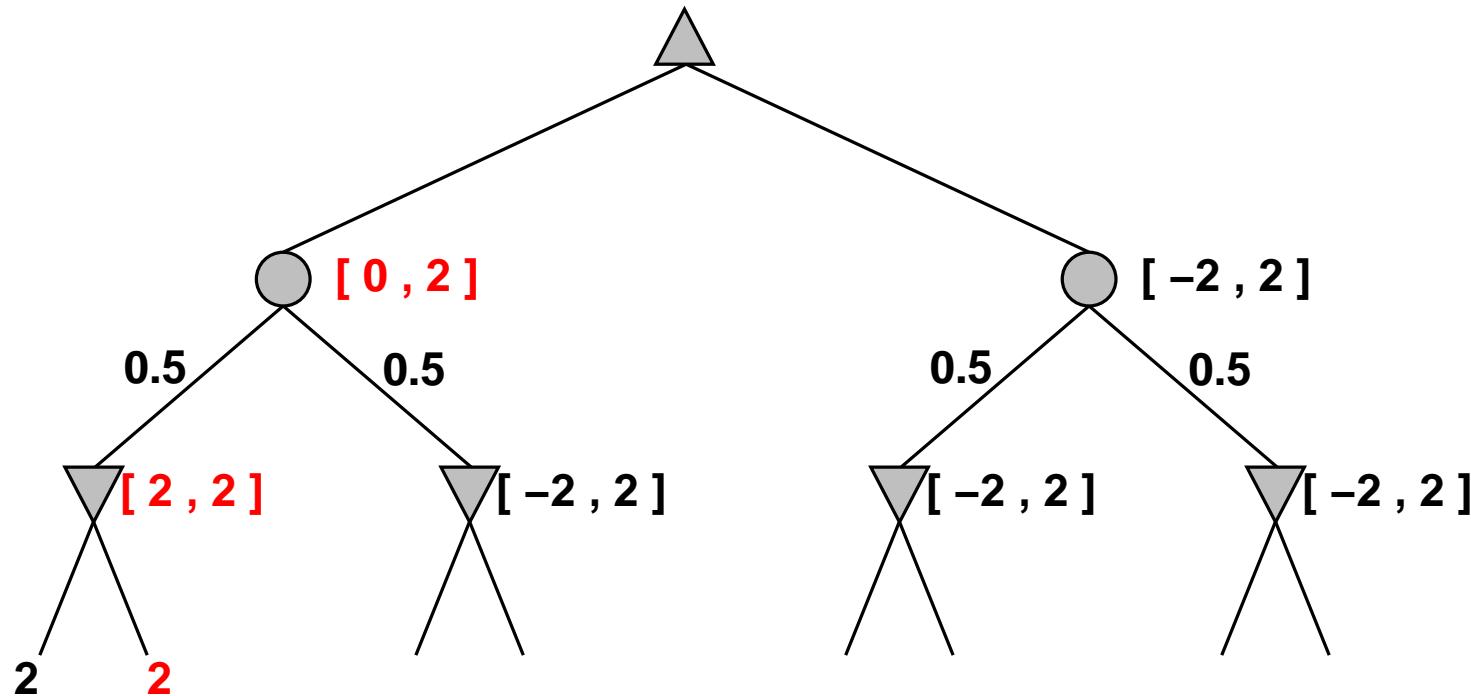
Strategia usrednionego minimax z odcieciem α - β

Bardziej efektywna, jeśli wartość wypłaty ograniczona



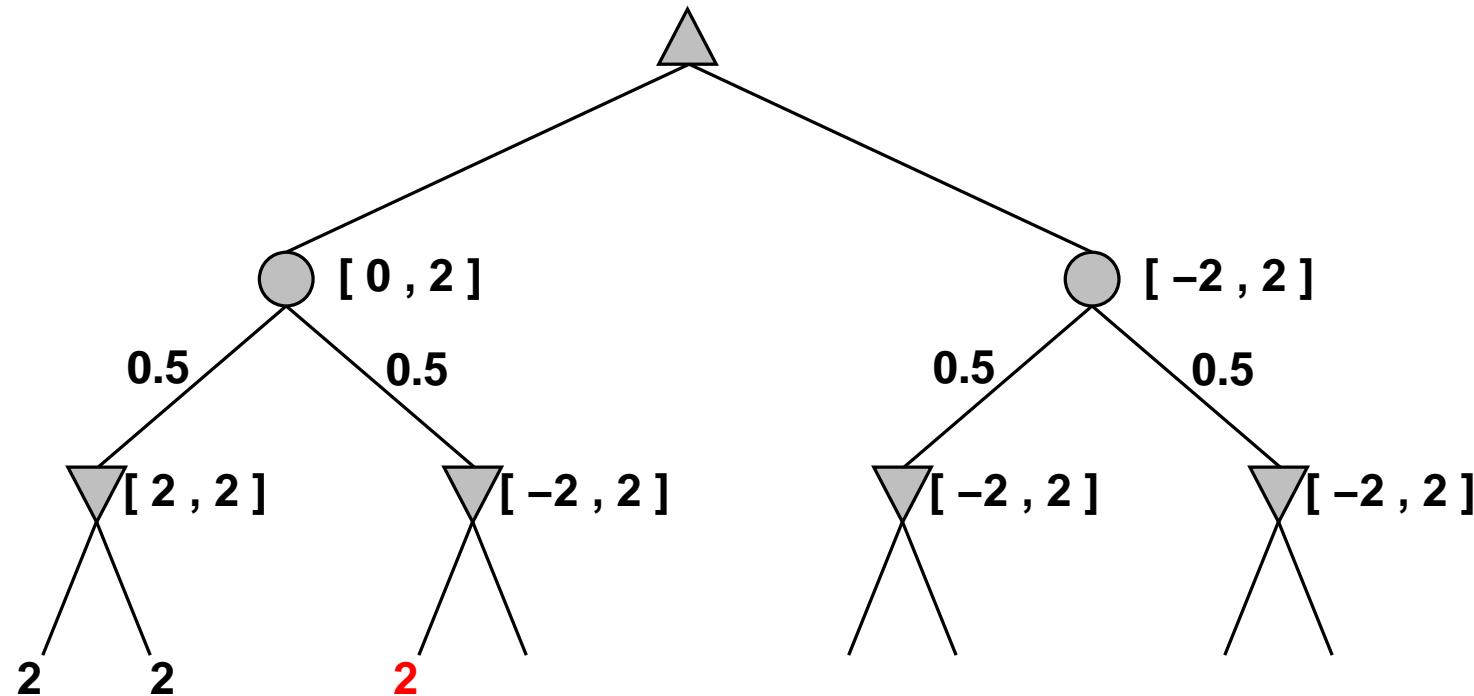
Strategia usrednionego minimax z odcieciem α - β

Bardziej efektywna, jeśli wartość wypłaty ograniczona



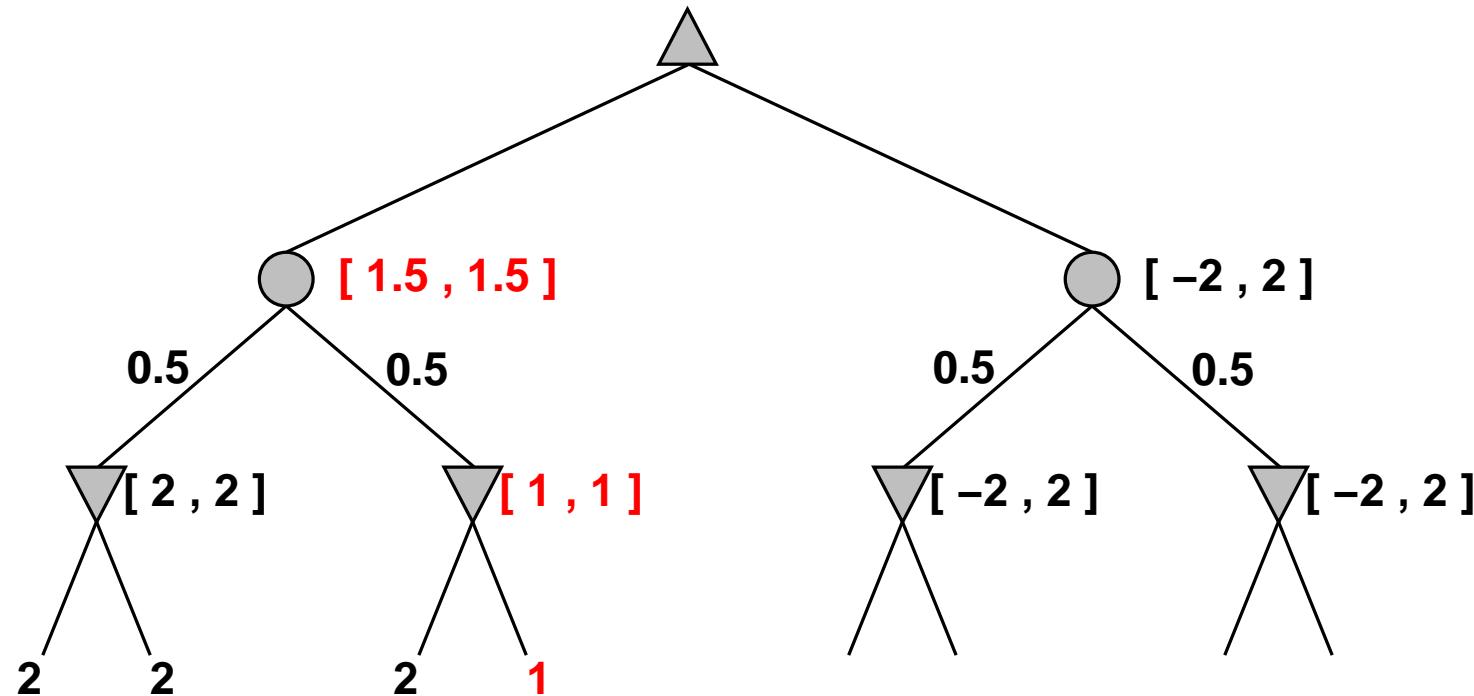
Strategia usrednionego minimax z odcieciem α - β

Bardziej efektywna, jeśli wartość wypłaty ograniczona



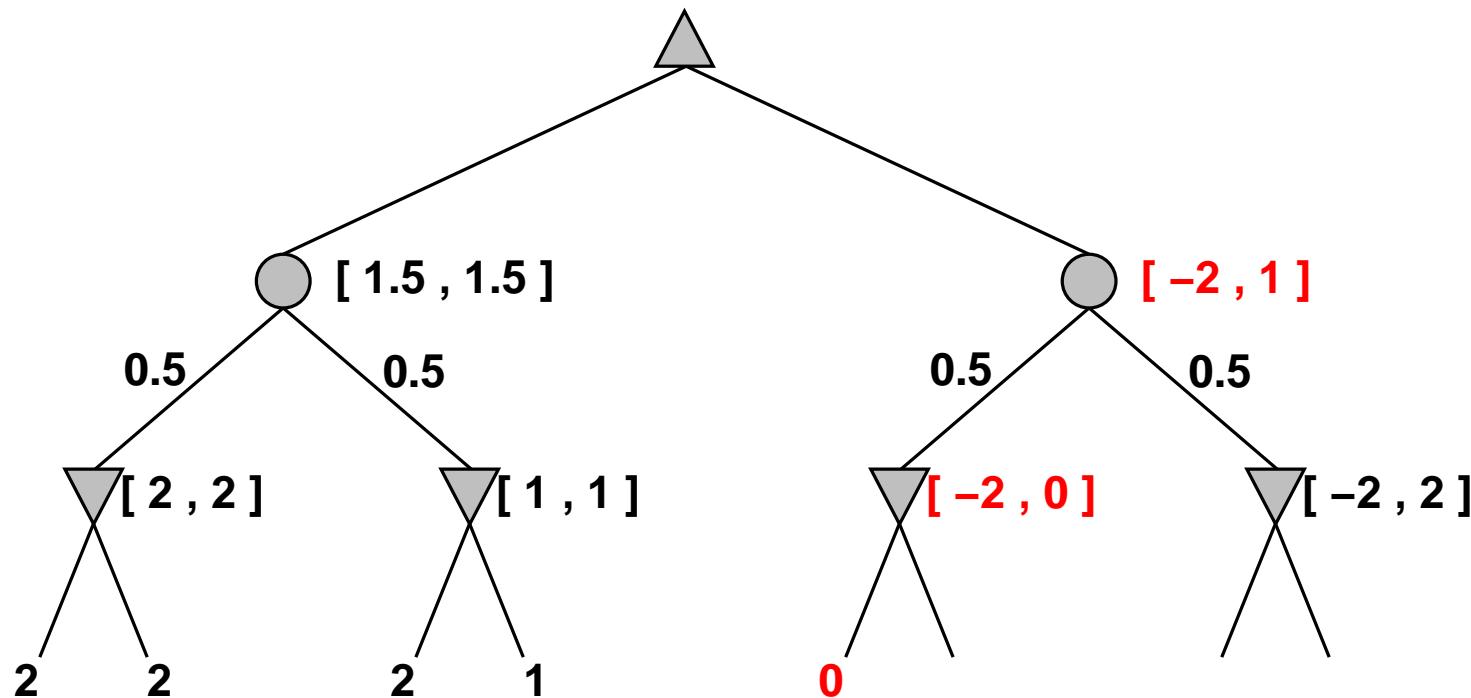
Strategia usrednionego minimax z odcieciem α - β

Bardziej efektywna, jeśli wartość wypłaty ograniczona



Strategia usrednionego minimax z odcieciem α - β

Bardziej efektywna, jeśli wartość wypłaty ograniczona



Gry niedeterministyczne: własności

Rzuty kostką zwiększą b : 21 możliwych rzutów dla 2 kostek

Backgammon ≈ 20 dopuszczalnych posunięć

$$\text{głębokość } 4 = 20 \times (21 \times 20)^3 \approx 1.2 \times 10^9$$

Jak głębokość wzrasta, prawdopodobieństwo osiągnięcia danego węzła maleje
 \Rightarrow wartość sprawdzania wprzód jest nikła

Odcinanie α - β jest dużo mniej efektywne

Program TDGAMMON:

przeszukiwanie na głębokość 2
+ bardzo dobra funkcja oceny stanu EVAL
 \approx poziom mistrza świata

Gry z niepełna informacją

Np. gry karciane, w których początkowy zestaw kart przeciwnika jest nieznany

Można policzyć prawdopodobieństwo każdego rozdania

⇒ wygląda jak jeden “duży” rzut kostką na początku gry

Pomysł:

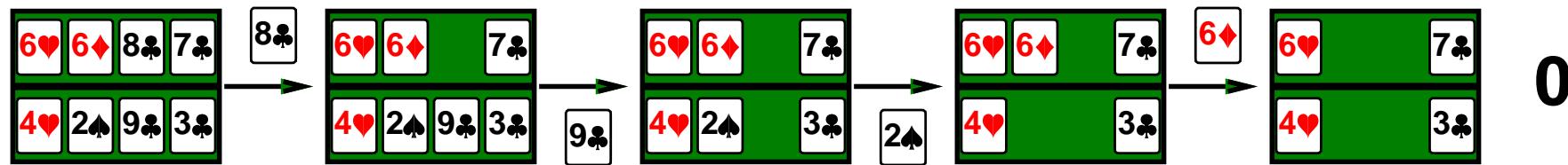
algorytm oblicza wartość minimax dla każdej akcji w każdym możliwym rozdaniu i wybiera akcje z największą wartością uśrednioną po wszystkich rozdaniach

GIB, najlepszy program do brydża, przybliża tą ideę

- 1) generuje 100 rozdań zgodnych z informacją z licytacji
- 2) wybiera akcję, która zbiera średnio najczęściej lew

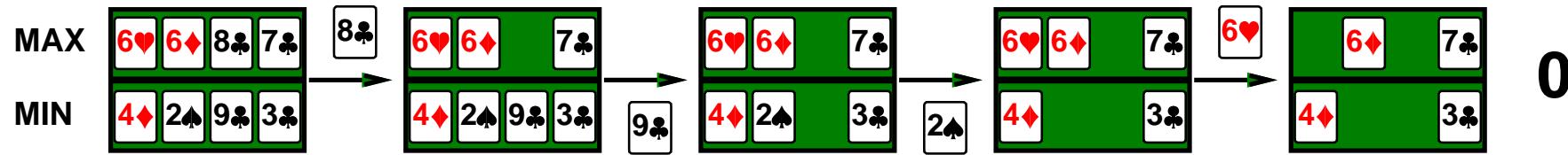
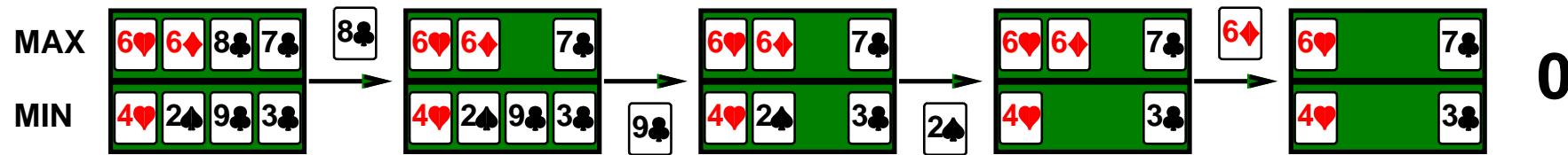
Gry z niepełna informacją: przykład

Gra w otwarte karty, MAX gra pierwszy



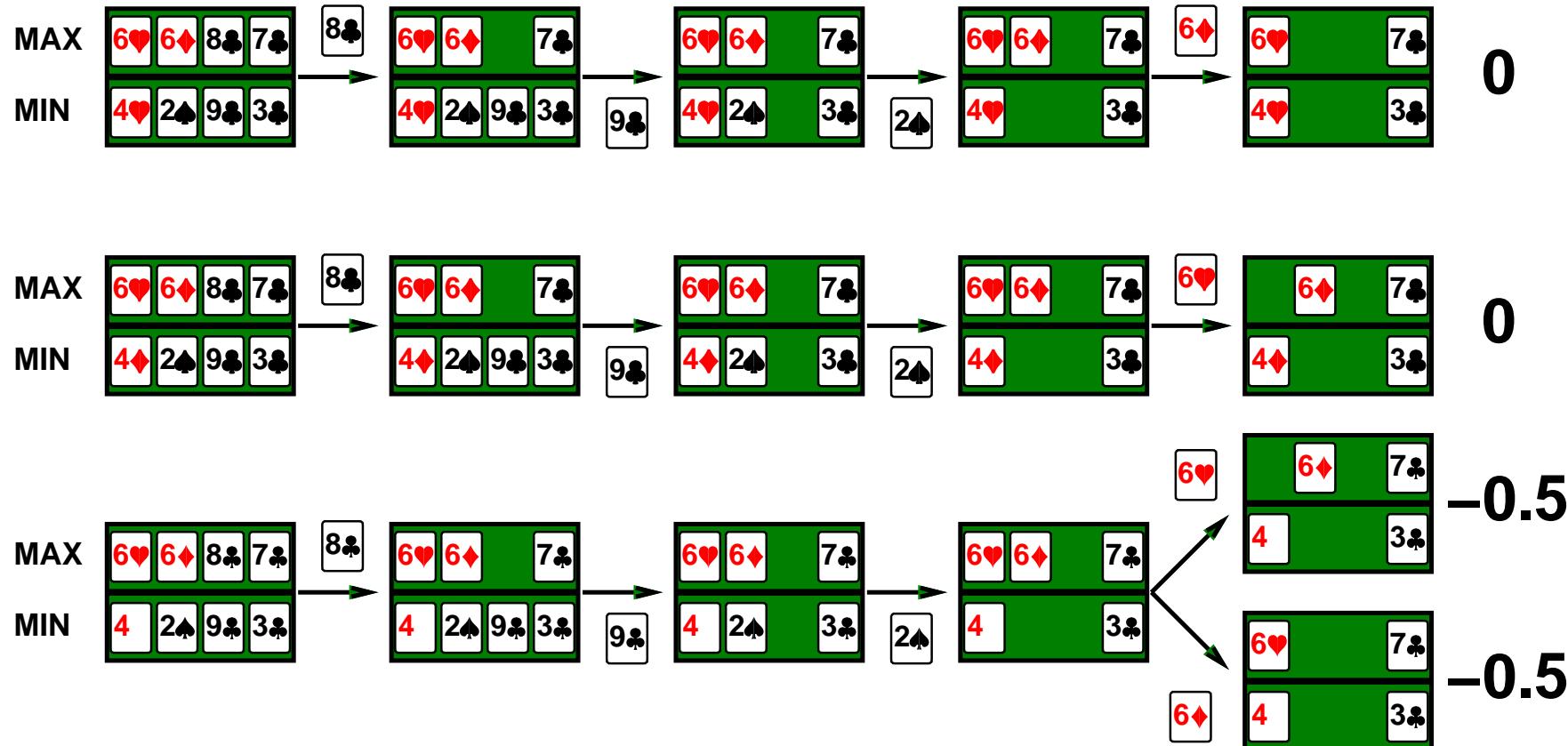
Gry z niepełna informacją: przykład

Gra w otwarte karty, MAX gra pierwszy



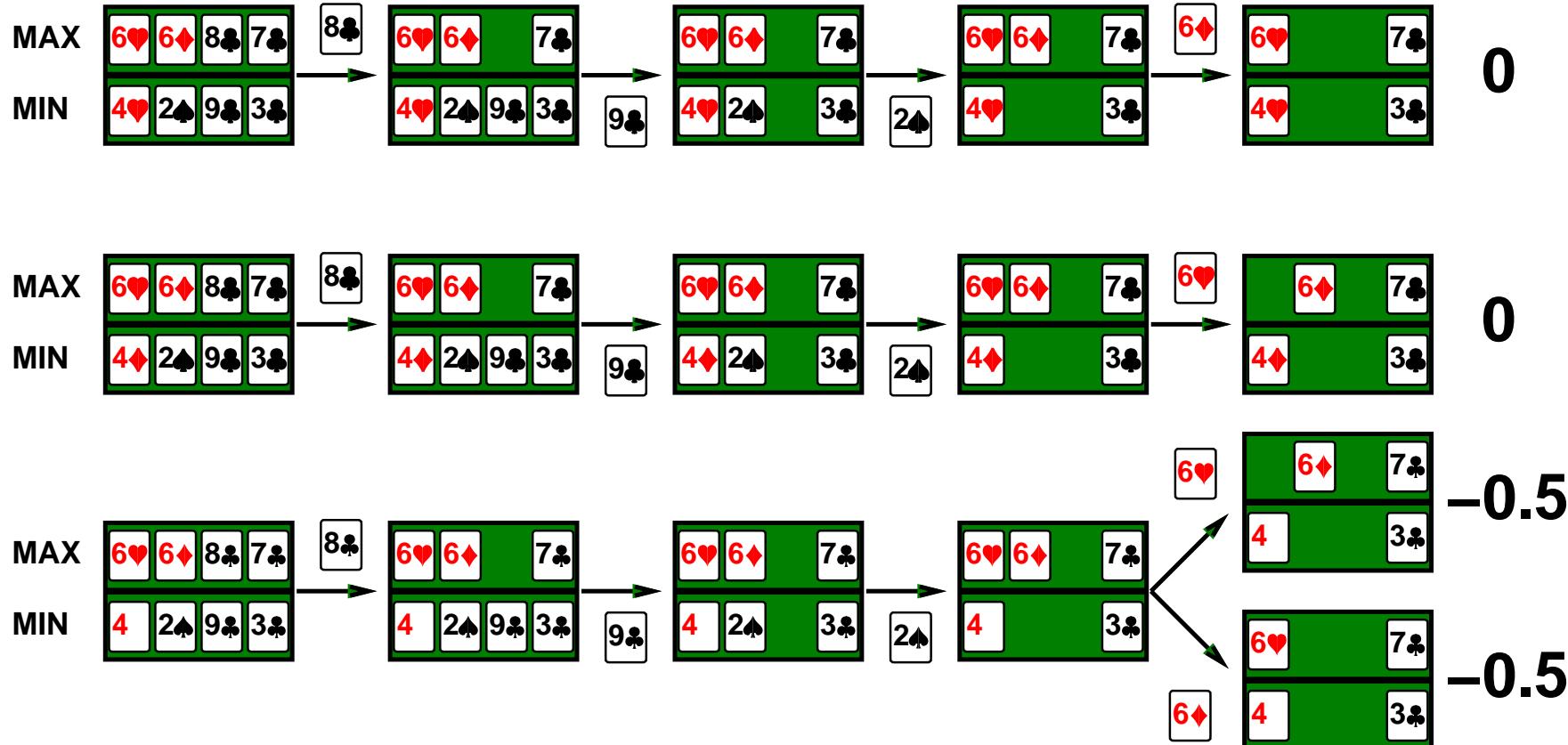
Gry z niepełna informacją: przykład

Gra w otwarte karty, MAX gra pierwszy



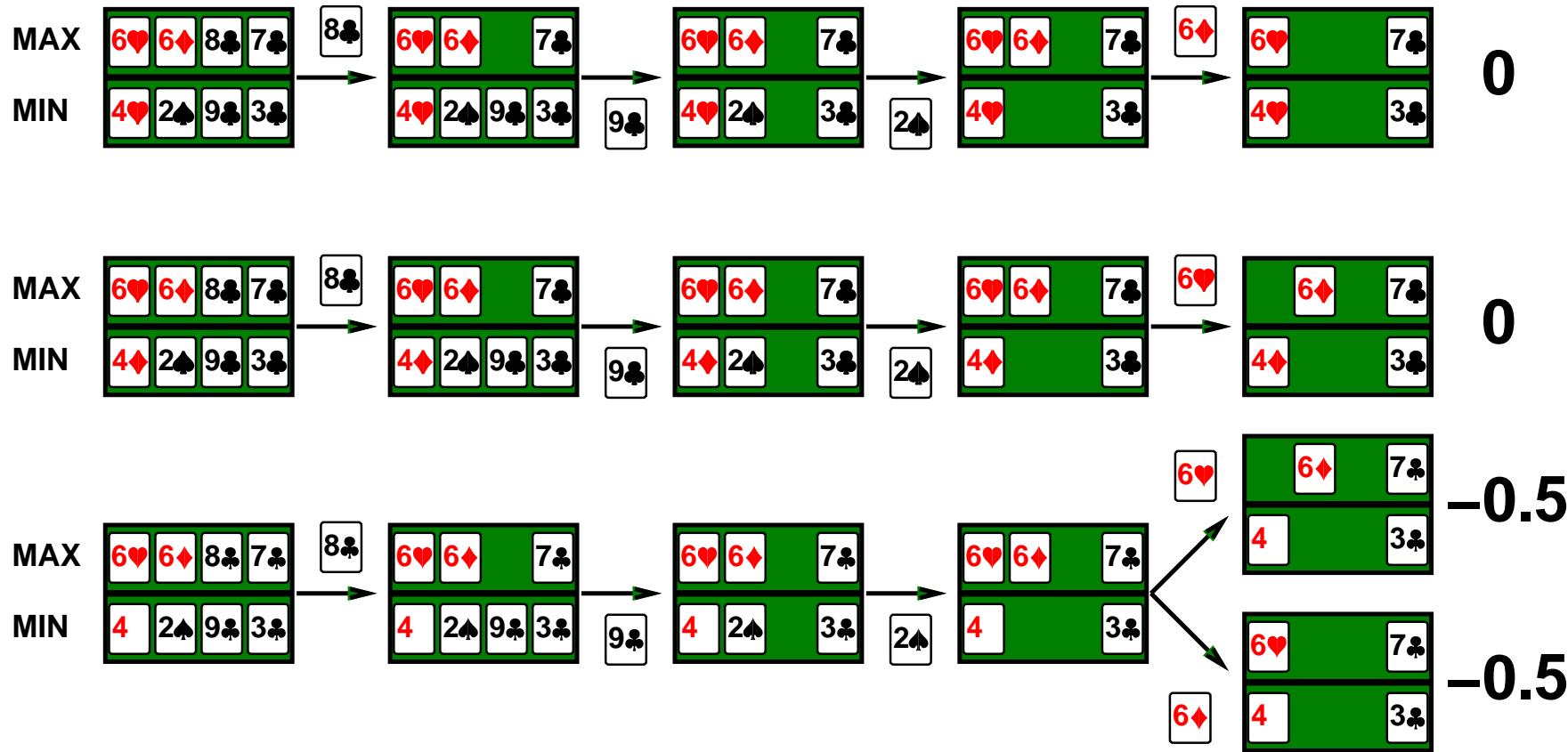
Gry z niepełna informacją: przykład

Gra w otwarte karty, MAX gra pierwszy



Jeśli MAX rozpocząłby z 6♥ lub 6♦, miałby gwarantowany remis

Gry z niepełna informacją: przykład



Jeśli MAX rozpocząłby z $6\heartsuit$ lub $6\diamondsuit$, miałby gwarantowany remis
 ⇒ uśrednienie nie jest optymalne!
 ⇒ ważne, która informacja będzie dostępna w którym momencie gry