Algorytmy

Norbert Jankowski

Katedra Informatyki Stosowanej

www.is.umk.pl/~norbert/algorytmy

KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI

Projekt współfinansowany ze środków
Uni Europejskiej w ramach
Europejskiego Funduszu Społecznego

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

Literatura I

A. V. Aho, J. E. Hopcroft, J. D. Ullman.
Projektowanie i analiza algorytmów.
Helion, Warszawa, 2003.

A. V. Aho, J. E. Hopcroft, J. D. Ullman.
Projektowanie i analiza algorytmów.
Państwowe Wydawnictwa Naukowe, Warszawa, 1983.

Niklaus Wirth.
Algorytmy + Struktury Danych = Programy.
Wydawnictwa Naukowo–Techniczne, Warszawa, wydanie 2, 1989.

T. H. Cormen, C. E. Leiserson, R. L Rivest.
Wprowadzenie do algorytmów.
Wydawnictwa Naukowo–Techniczne, Warszawa, 1997.

Literatura II

A. V. Aho, J. D. Ullman.
Wykłady z informatyki z przykładami w języku C.
Helion, Warszawa, 2003.

L. Banachowski, K. Diks, W. Rytter.
Algorytmy i struktury danych.
Wydawnictwa Naukowo–Techniczne, Warszawa, 1996.

M. M. Sysło, N. Deo, J. Kowalik.
Algorytmy optymalizacji dyskretnej.
Państwowe Wydawnictwa Naukowe, Warszawa, 1993.

D. Harel.
Rzecz o istocie informatyki. Algorytmika.
Wydawnictwa Naukowo–Techniczne, Warszawa, 1992.

Literatura III

L. Banachowski, A. Kreczmar, W. Rytter.
Analiza algorytmów i struktur danych.
Wydawnictwa Naukowo–Techniczne, Warszawa, 1989.

L. Banachowski, A. Kreczmar.
Elementy analizy algorytmów.
Wydawnictwa Naukowo–Techniczne, Warszawa, 1982.

D. E. Knuth.
Sztuka Programowania, wolumen I–III.
Wydawnictwa Naukowo–Techniczne, 2002.

Algorytmy z powrotami

- Automatyczne rozwiązywanie problemów, gdzie nie ma (prostych) rozwiązań analitycznych.
- Podział na pod-zadania + metoda prób i błędów.
- Bardzo często rekurencja.
- Niestety najczęściej złożoność eksponencjalna ...

Drogi skoczka szachowego

	3		2	
4				1
		S		
5				8
	6		7	

- Skoczek (koń) porusza się zgodnie z regułami gry w szachy
- Start z ustalonej pozycji
- Zadanie odwiedzić każdą pozycję dokładnie jeden raz

Zagadnienie można zamienić na próbę wykonania kolejnego ruchu skoczkiem, jeśli to tylko możliwe.

```
1  PróbujNastępnyRuch()
2  {
3      zapoczątkuj wybieranie ruchów;
4      do {
5          wybierz następnego kandydata z listy ruchów;
6          if (dopuszczalny)
7          { zapisz ruch
8              if (na szachownicy wolne pola)
9              { PróbujNastępnyRuch
10                 if (nieudany) wykreśl ostatni ruch
11             }
12         }
13     } while ( ruch był nie udany && jest następny kandydat);
14 }
```

- Zapamiętujmy w macierzy h numery kroków wydawanych przez skoczka

```
1    var h : array [1..n,1..n] of integer;
```

- $h[x, y] = 0$ — pole nieodwiedzone
- $h[x, y] = i$ — pole odwiedzone w i-tym kroku
- Musimy mieć możliwość przekazywania informacji o zewnętrznych wywołaniach. *Czy ruch się powiódł*? Będziemy to robić poprzez zmienną q (true=sukces)
- *Czy ruch jest dopuszczalny?* $= 1 \leq u, v \leq n$ && $h[u, v] = 0$, u,v - nowa pozycja skoczka
- *Czy są wolne pola?* $= i < n^2$

```
1  Próbuj(int i, x , y, bool& q) {
2  int u,v;
3  { zapoczątkuj wybieranie ruchów;
4      do { u, v współrzędne następnego ruchu;
5          if (1<=u<=n) && (1<=v<=n) && (h[u,v]=0)
6          { h[u,v]=i;
7              if (i<n*n )
8              { Próbuj(i+1, u,v, q);
9                  if ( !q) h[u,v]=0;
10             }
11             else q=true;
12         }
13     } while( !q && jest następny kandydat);
14 }
```

	3		2	
4				1
		S		
5				8
	6		7	

```
a[1]= 2;  b[1]= 1;
a[2]= 1;  b[2]= 2;
a[3]=−1;  b[3]= 2;
a[4]=−2;  b[4]= 1;
a[5]=−2;  b[5]=−1;
a[6]=−1;  b[6]=−2;
a[7]= 1;  b[7]=−2;
a[8]= 2;  b[8]=−1;
```

```
1  Próbuj(int i, x,y, bool& q) {
2  int u,v,k;
3  { k=0;
4      do { k=k+1;
5           u=x+a[k]; v=y+b[k];
6           if ((1<=u<=n) && (1<=v<=n) && (h[u,v]==0))
7           { h[u,v]=i;
8               if (i<n*n)
9               { Próbuj(i+1, u,v, q);
10                  if (! q) h[u,v]=0;
11              } else q=true;
12          }
13     } while(! q && jest następny kandydat);
14 }
```

```
skoczek(){
 ...
{
    a[1]= 2;  b[1]= 1;
     ...
    for  i=1 to n do
        for  j=1 to n do
            h[i,j]=0;
    h[1,1]=1;
    Próbuj(2,1,1,q);
    if  (q )
        drukuj  macierz  h
    else  printf('brak␣rozwiązania');
}
```

Wynik dla szachownicy 5x5

1	6	15	10	21
14	9	20	5	16
19	2	7	22	11
8	13	24	17	4
25	18	3	12	23

```
Próbuj−OGÓLNY−SZKIC()
{
    zapoczątkuj wybieranie kandydatów;
    do{
        wybierz następnego kandydata;
        if (dopuszczalny)
        { zapisz go;
            if (rozwiązanie niepełne)
            { Próbuj−OGÓLNY−SZKIC;
                if (próba nieudana) usuń zapis
            }
        }
    } while( próba nieudana && jest następny kandydat);
}
```

Problem ośmiu hetmanów

- Na szachownicy umieścić 8 hetmanów, ale tak aby żaden nie szachował żadnego innego.
- Rozwiązanie: analitycznie — NIE, rekurencja+powroty — TAK
- Podział problemu znów na próby i powroty

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

```
Próbuj8H(int i)
{
    zapoczątkuj wybieranie pozycji i-tego hetmana;
    do {
        dokonaj następnego wyboru pozycji;
        if (bezpieczna)
        { ustaw hetmana;
            if (i<8)
            { Próbuj8H(i+1);
                if (próba nieudana) usuń hetmana;
            }
        }
    } while(próba nieudana && jest więcej pozycji);
}
```

Szach gdy:

- w tej samej kolumnie
- w tym samym wierszu
- na tej samej przekątnej(!)

Reprezentacja:

```
int x[8]; { pozycja w i-tej kolumnie }
bool a[8];
    { a[j] brak hetmana w j-tym wierszu }
bool b[15];
    { b[k] brak hetmana na k-tej przekątnej ↙ }
bool c[15];
    { c[k+7] brak hetmana na k-tej przekątnej ↘ }
```

Ustaw hetmana w pozycji: i-ta kolumna, j-ty wiersz:

```
x[i]=j; a[j]=false; b[i+j]=false; c[i-j+7]=false;
```

Usuń hetmana z pozycji i,j:

```
a[j]=true; b[i+j]=true; c[i-j+7]=true;
```

Warunek *bezpieczna pozycja* i,j:

```
a[j] && b[i+j] && c[i-j+7]
```

```
Próbuj8H(int i, bool& q) {
int j;
{ j=-1;
    do { j=j+1; q=false;
        if (a[j] && b[i+j] && c[i-j+7])
        { x[i]=j;
            a[j]=false; b[i+j]=false; c[i-j+7]=false;
            if( i<7)
            { Próbuj8H(i+1, q);
                if (! q) {
                    a[j]=true; b[i+j]=true; c[i-j+7]=true;
                }
            } else q=true;
        }
    } while( !q && (j!=7));
}
```

Przeszukiwanie „*najpierw najlepszy*” i „*wiązką*”

- Nie zawsze (wręcz b. rzadko) można pozwolić sobie na tak dokładne przeszukanie przestrzeni rozwiązań jak dla problemu skoczka bądź 8 hetmanów.
- Powstały metody, które co prawda nie służą do znajdowania dokładanych (optymalnych) rozwiązań, jak jest to konieczne w nie których problemach, ale potrafią za to znaleźć *podoptymalne* rozwiązania, co w wielu zastosowaniach wystarcza (np. gry strategiczne).
- Takie algorytmy korzystają z niemal identycznej struktury algorytmu, lecz nie prowadzą eksploracji wszystkich *kandydatów* lecz ich podzbiór, który jest wyznaczany odpowiednimi kryteriami zależnymi od problemu.
- Jakość kryteriów określających zbiór kandydatów ma kluczowe znaczenie dla powodzenia algorytmu.

```
Szukaj−NN()
{
    zapoczątkuj wybieranie kandydatów;
    wybierz najlepszego dopuszczalnego kandydata;
    zapisz go;
     if (rozwiązanie niepełne)
     {
         Szukaj−NN;
         if (próba nieudana) usuń zapis
     }
}
```

```
Szukaj−Wiązką()
{
    zapoczątkuj wybieranie kandydatów;
    wybierz podzbiór (wiązkę) najlepszych kandydatów;
    do {
        weź następnego kandydata;
        if (dopuszczalny)
        { zapisz go;
            if rozwiązanie niepełne then
            { Szukaj−Wiązką;
                if (próba nieudana) usuń zapis
            }
            else zapisz rozwiązanie;
        }
    } while( ! podzbiór kandydatów przeszukany);
}
```

α, β-obcięcie

Inna wersja: α, β-obcięcie — gry strategiczne, poziomy obcięcia dla funkcji oceny ruchu swojego i przeciwnika.

```
Szukaj−AB(i, rola)
{
  s = wszystkie ruchy;
  s = s / niedopuszczalne;
  foeach (s_j in S)
  {
    nowyStan = staryStan + ruch s_j;
    f_j = F(nowyStan, rola)
  }
  s = s / {s_j : f_j < AB};
  do {
    zrób ruch + zapisz go;
    if (rozwiązanie niepełne && i < maxGłębokość)
      {
        Szukaj−AB(i+1, !rola);
        usuń ruch;
      }
    else zapisz rozwiązanie (częściowe);
  } while (podzbiór kandydatów nieprzeszukany);
}
```

Operacje na zbiorach

Member(a,S) — Czy a należy do S.

Insert(a,S) — $S = S \cup \{a\}$

Delete(a,S) — $S = S \setminus \{a\}$

Union(A,B,C) — Zastąp zbiory A i B przez C.

Find(a) — Podaje nazwę zbioru do którego należy a.

Split(a,S) — Rozdziela S na $S_1 = \{b : b \leq a \wedge b \in S\}$ i $S_2 = \{b : b > a \wedge b \in S\}$.

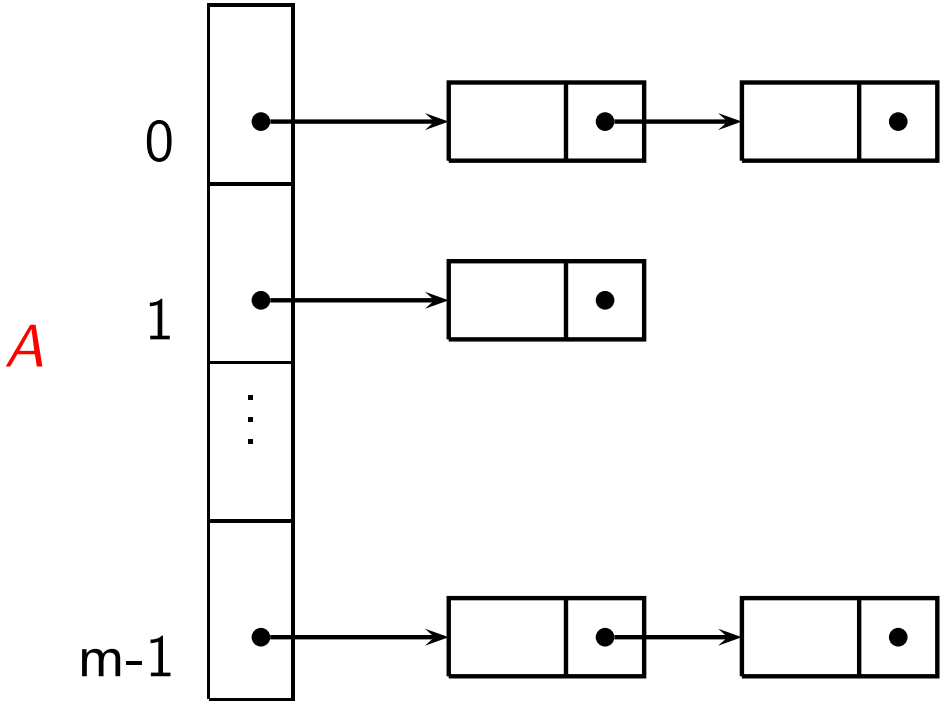
Min(S) — Podaje wartość elementu minimalnego.

- Głównie interesuje nas wykonywanie ciągów instrukcji typu: *Member*, *Insert*, *Delete*
- Zastosowania: słownikowe, wiele wiele operacji jedno- i wielo-zbiorowych (np. algorytm Kruskala znajdowania minimalnego drzewa rozpinającego).
- Dlatego: głównie będzie interesowała nas złożoność nie pojedynczej instrukcji lecz całego ciągu instrukcji σ.

Haszowanie

- Rozważmy instrukcje: Insert, Delete, Member.
- Niech S będzie pewnym dużym zbiorem.

- Haszowanie: umieszcza elementy zbioru S na odpowiedniej liście w tablicy list:



- Funkcja haszująca $h(a) = i$ wyznacza, na której liście tablicy haszującej A ma znaleźć się element a.

- Wykonanie $Insert(a, S)$ to:
  – wyznaczenie h(a),
  – przeszukanie A[h(a)],
  – jeśli ta lista nie zawiera a, należy na koniec listy dopisać a
- Wykonanie $Delete(a, S)$ to:
  – wyznaczenie h(a),
  – przeszukanie A[h(a)],
  – usunięcie a z listy A[h(a)] (jeśli tylko jest na liście).
- Wykonanie $Member(a, S)$ — analogicznie

Złożoność obliczeniowa:

Najgorszy przypadek: Może się zdarzyć, że dla różnych n instrukcji *Insert* funkcja haszująca $h(a)$ da taką samą wartość i, wtedy (niestety) wszystkie poszczególne elementy a znajdą się na tej samej liście tablicy haszującej $A[h(a)]$. Oznacza to, że wykonanie n instrukcji *Insert* ma złożoność $O(n^2)$.

Czas oczekiwany haszowania: Założenie, że wartości $h(a)$ są jednakowo prawdopodobne i że wstawiamy $n \leq m$ wartości, to dla i-tego wstawiania oczekiwana długość listy $A[h(a)]$ wynosi $(i-1)/m$ $(< 1!)$.
Co prowadzi do złożoności $O(n)$ dla n instrukcji *Insert*.

Ogólnie oczekiwany koszt 1 operacji: wynosi $O(1+\alpha)$, gdzie $\alpha = \frac{n}{m}$.

Funkcja haszująca $h(a)$:

- Ważnym założeniem jest, że $h(a)$ da się obliczyć w $O(1)$.
- Wybór funkcji haszującej jest kluczowy dla późniejszej pracy algorytmu haszowania.
- Ogólnie funkcja haszująca powinna każdy klucz z jednakowym prawdopodobieństwem odwzorowywać w jedną zwartości m:

$$\sum_{a\,:\,h(a)=j} P(a) = \frac{1}{m} \qquad \text{dla każdego } j = 0, \ldots, m-1$$

Jest to warunek prostego równomiernego haszowania.

- Gdy założyć, że wartości a są jednakowo prawdopodobne i ograniczone, np. $[0, N]$, gdzie $N \gg m$, to jako funkcji haszującej można użyć operatora modulo:

$$h(a) = a \ \% \ m$$

- Z kolei gdy klucze a będą losowymi wartościami równomiernie rozłożonymi na przedziale $0 \leq a < 1$, to funkcja haszująca może przyjąć postać:

$$h(a) = \lfloor am \rfloor$$

- Haszowanie przez mnożenie:

$$h(a) = \lfloor m(aB \ \% \ 1) \rfloor$$

najpierw klucz a (całkowite wartości) mnożymy przez pewną stałą ($0 \leq B < 1$) z czego pobieramy część ułamkową, a następnie mnożymy przez rozmiar tablicy A (m) i zaokrąglamy do całości. Dobre $B \approx (\sqrt{5} - 1)/2 = 0,6180339887\ldots$

- Czasami należy skorzystać z pewnych własności heurystycznych przy doborze funkcji haszującej – szczególnie gdy nie znamy rozkładu P.
- Dla podobnych napisów f. haszująca powinna zapewnić dobry rozkład – różne wartości $h(a)$ – dla podobnych napisów, np. tablica1, tablicaX. Tu także zastosowanie mogą mieć funkcje modularne.

Co gdy $n = m$?

Gdy liczba przechowywanych elementów w danym zbiorze S osiągnie wartość bliską (równą lub większą) długości tablicy A należy zwiększyć pojemność tablicy haszującej:

- Utworzyć nową tablicę A_2 długości $2m$.
- Przez powtórne haszowanie dokonujemy kopiowania elementów $A = A_1$ do A_2
- Usuwamy A_1

Gdy z kolei dojdzie do przepełnienia A_2 należy postąpić analogicznie. . . Odwrotnie postępujemy, gdy liczba elementów w tablicy zmniejszyła się za bardzo.

Haszowanie otwarte

- Tablica M jest zwykłą tablicą elementów, które są przechowywane w zbiorze.
- Nie ma więc list ani innych wskaźników.
- Takie założenia prowadzą jednak do kolizji, które muszą być eliminowane. Robi się to przez powtórne haszowanie.

M[i]=k	normalna wartość
M[i]=∞	puste miejsce
M[i]=$-\infty$	miejsce puste po el. usuniętym

```
1  int HashInsert(k)
2  {
3      i=0;
4      while( i != m ){
5          j = h(k,i);
6          if( M[j] == ∞ || M[j] == −∞ ){
7              M[j]=k;
8              size++;
9              return j;
10         } else
11             i++;
12     }
13     error(" przepełnienie ␣ tablicy ␣/␣ reorganizacja ");
14 }
```

UWAGA: Tą i kolejne funkcje trzeba rozbudować o mechanizm rozszerzania i kurczenia tablicy M. Jednak próg nie może być blisko pełnej tablicy(!).

```
1  int HashSearch(k)
2  {
3      i=0;
4      while( i != m ) {
5          j = h(k,i);
6          if( M[j] == ∞ )
7              return −1;
8          if( M[j] == k )
9              return j;
10         else
11             i++;
12     }
13 }
```

```
int HashDelete(k)
{
    i=0;
    while( i!=m ) {
        j = h(k,i);
        if( M[j] == ∞)
            return −1;
        if( M[j] == k ) {
            M[j] = −∞;
            size−−;
            return j;
        } else i++;
    }
}
```

Funkcja haszujące dla haszowania otwartego

$$h(k,i) = (h_1(k) + ih_2(k)) \mod m$$

$$h_1(k) = k \mod m$$

$$h_2(k) = 1 + (k \mod m')$$

np. $m' = m - 1$

Drzewa przeszukiwań binarnych dla operacji zbiorowych

- Drzewa binarne.
  n instrukcji ($Insert, Delete, Member$) — $O(n^2)$.
  Średnio — $O(n \log n)$.
- Drzewa zbalansowane (drzewa czerwono–czarne, AVL) — n operacji ($Insert, Delete, Member$) dają złożoność $O(n \log n)$.

Prosto i wolno ...

Elementy zbioru: liczby całkowite od 1 do n.
Wykorzystuje: operacje Union i Find.

Niech R będzie tablicą i $R[i]$ = indeks zbioru.
Nazwy zbioru są nieistotne, przyjmujemy: $R[i] \in [1, n]$ (ekstremalnie każdy element należy do innego zbioru).

$1 = \{1, 3, 4, 6\}, \quad 2 = \{5\}, \quad 3 = \{2, 7\}$

	R
1	1
2	3
3	1
4	1
5	2
6	1
7	3

```
Union(R,A,B,C)
{
    for (int i=1; i<=n; i++)
        if ((R[i]==A) || (R[i]==B) )
            R[i]=C;
}
```

Koszt jednej operacji *Union* to $O(n)$, czyli ciąg n operacji oznacza złożoność $O(n^2)$.

Listy z dowiązaniami

Łatwo jednak można usprawnić powyższy algorytm:

- Użyjemy list z dowiązaniami – unikniemy przeglądania całej tablicy.
- Będziemy dołączać mniejszy zbiór do większego – unikniemy przeglądania dłuższego ze zbiorów.

Teraz oprócz tablicy R będziemy mieli także inne tablice.

LIST[A] – lista z dowiązaniami. $LIST[A]$ wskazuje indeks pierwszego elementu zbioru A w R.

NEXT[i] – daje indeks kolejnego elementu tego samego zbioru co element $R[i]$. Niech $j = LIST[A]$, wtedy $NEXT[j]$ to drugi element zbioru A, $NEXT[NEXT[j]]$ to trzeci, etc.

SIZE[A] – daje bieżącą liczbę elementów zbioru A.

EXTERNAL_NAME[A] – to zewnętrzna nazwa zbioru A (R[i] – daje wewnętrzną nazwę).

Przykład

Mamy zbiory: $1 = \{1, 3, 5, 7\}$, $2 = \{2, 4, 8\}$, $3 = \{6\}$

	R	NEXT
1	2	3
2	3	4
3	2	5
4	3	8
5	2	7
6	1	0
7	2	0
8	3	0

	LIST	SIZE	EXT_NAME
1	6	1	3
2	1	4	1
3	2	3	2

	INT_NAME
1	2
2	3
3	1

```
Union2(I,J,K)
{    A=INT_NAME[I]; B=INT_NAME[J];
     { dopisujemy krótszą do dłuższej }
     if (SIZE[A]>SIZE[B]) zamieniamy rolami A i B;
     e = LIST[A];
     while (e!=0) { przepisywanie elementów z A do B }
     {
          R[e]=B;
          last =e;
          e=NEXT[e];
     }
     NEXT[last]=LIST[B];
     LIST[B]=LIST[A];
     SIZE[B]=SIZE[A]+SIZE[B];
     INT_NAME[K]=B; EXT_NAME[B]=K;
}
```

Złożoność *Union*2

Złożoność wykonania $n-1$ instrukcji *Union*2 to $O(n \log n)$.

Taka złożoność wynika z faktu, że każdy element może być przełożony tylko do zbioru niemniej niż 2 razy większego.

Co oznacza, że każdy element może być przenoszony co najwyżej $\log n$ razy, daje to koszt $O(\log n)$ (dla pojedynczego elementu).

Złożoność pojedynczego $Find(i)$ pozostaje $O(1)$.

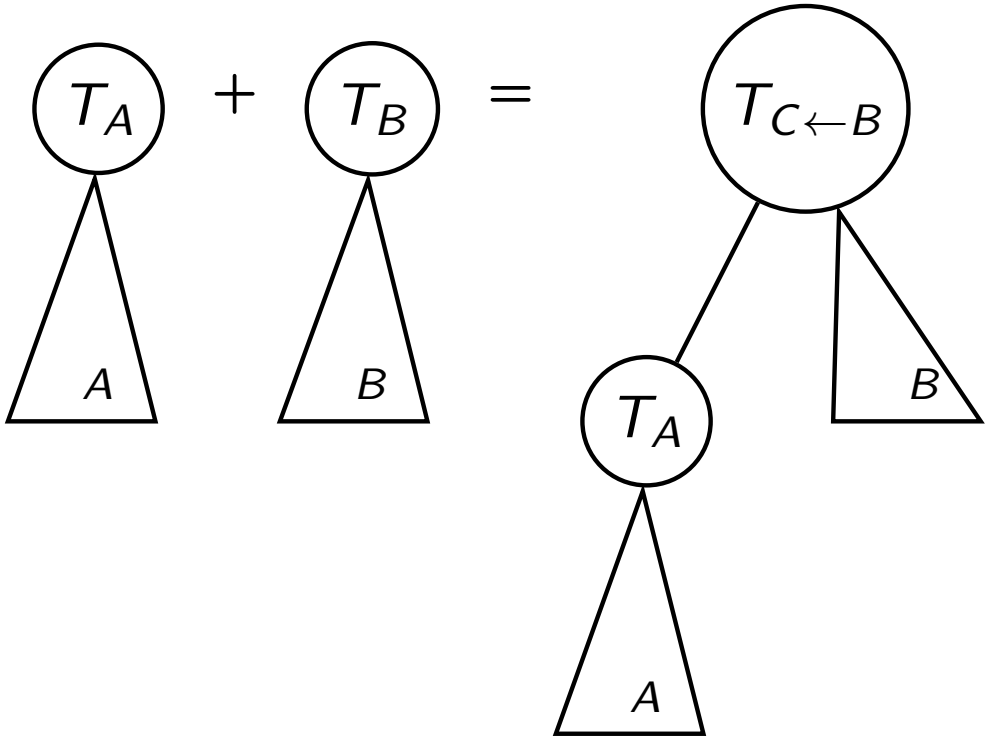
Struktury drzew dla problemu Union-Find

- Rodzina zbiorów będzie reprezentowana przez las drzew. Jedno drzewo = jeden zbiór.
- Zbiór A ma (tylko) korzeń T_A.
- Nazwę zbioru ma korzeń drzewa.

- Union(A,B,C) można zrealizować poprzez:

– uczynienie korzenia T_A synem T_B,

– zmianę nazwy korzenia T_B na C.



Koszt łączenia: $O(1)$

- Find(i) można zrealizować poprzez:

– odszukanie wierzchołka, który odpowiada i w pewnym drzewie T, można użyć tablicy wskaźników na węzły drzewa: t[i] — wsk. na i-ty element

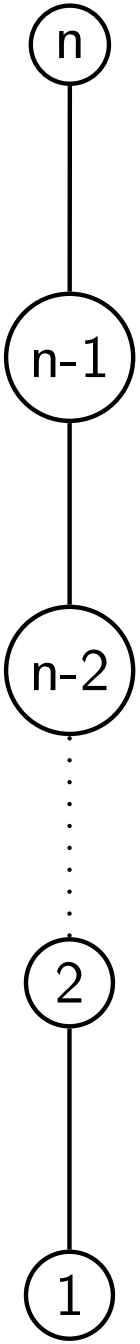
– przejście z wierzchołka do korzenia, w którym zapisana jest nazwa drzewa do którego należy i.

Koszt Find: $O(h)$, gdzie h, to wysokość drzewa

Koszt Find: jest rzędu długości ścieżki z wierzchołka i do korzenia drzewa.

Rozpatrzmy ciąg instrukcji:

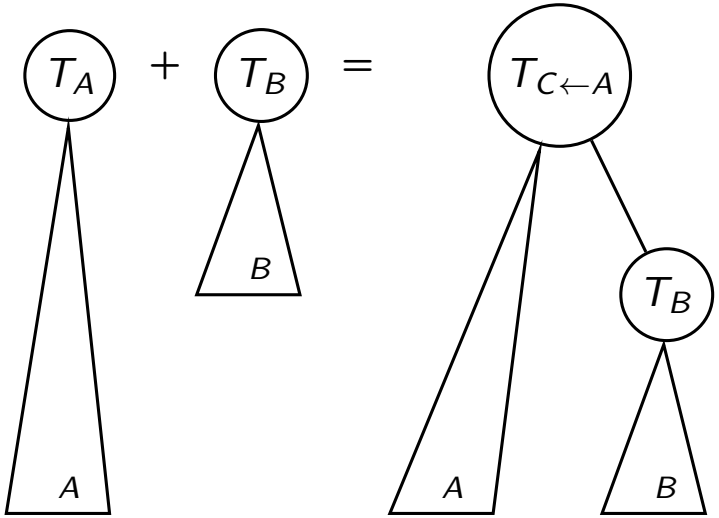
Union(1,2,2)
Union(2,3,3)
⋮
Union(n-1,n,n)
Find(1)
Find(2)
⋮
Find(n)



Ciąg instrukcji Union powoduje powstanie drzewa listy:
Wtedy koszt wykonania n instrukcji Find wynosi:

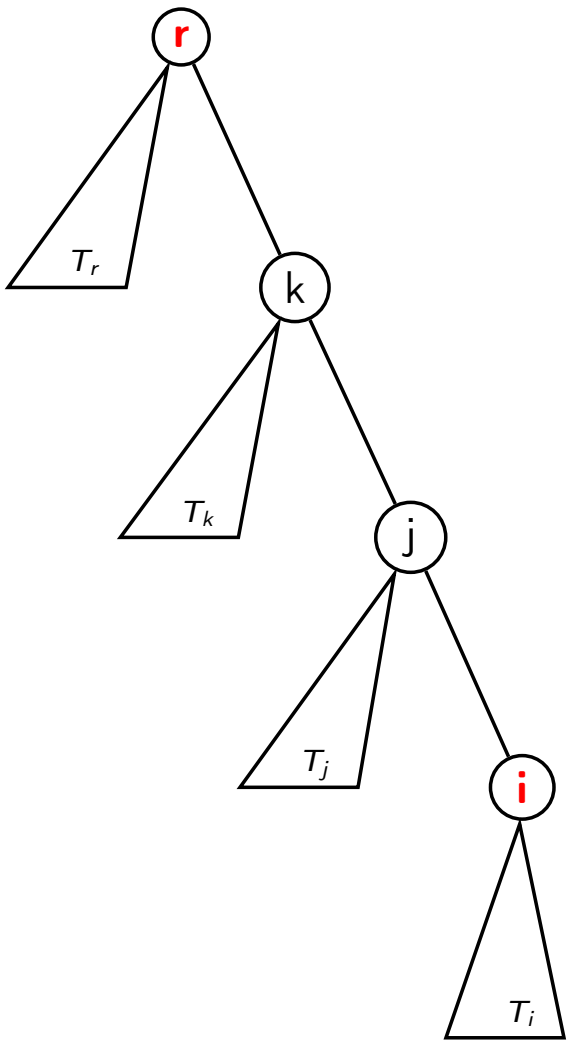
$$\sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}.$$

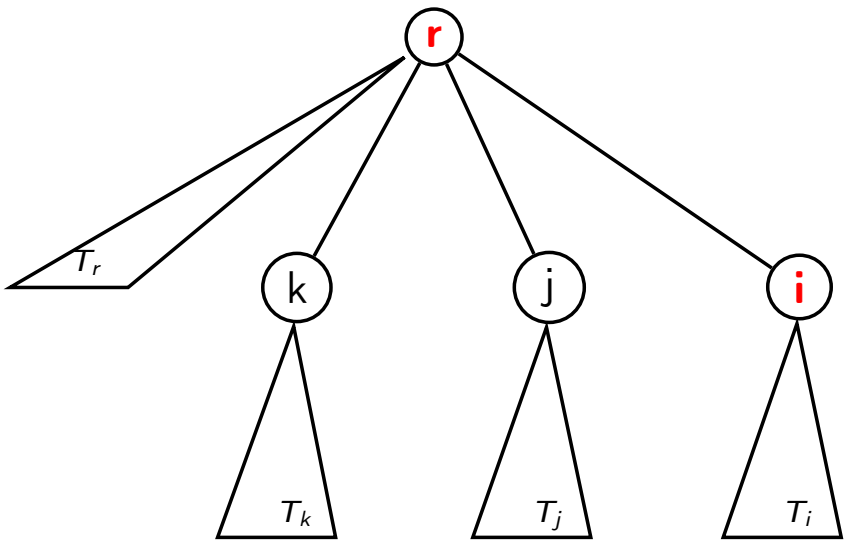
Taki koszt wymusza, aby łącząc minimalizować wysokość wynikowego drzewa poprzez przyłączanie mniejszego drzewa pod większe.



Kompresja ścieżki

- Kolejna modyfikacja polega na kompresji ścieżki podczas wykonywania funkcji Find — koszt tej funkcji jest dużo większy niż Union.
- Koszt Find(i) jest związany z długością ścieżki od wierzchołka i do korzenia r drzewa.
- $i, v_1, v_2, \ldots, v_{n-1}, r$ — ścieżka od i do r.
- Kompresja ścieżki to uczynienie każdego wierzchołka v_j synem korzenia r:


r
T_r
k
T_k
j
T_j
i
T_i


r
T_r
k
T_k
j
T_j
i
T_i

Implementacja może być oparta o tablice.
Inicjowanie:
$COUNT[i] = 1 \qquad 1 \leq i \leq n$ — liczność zbioru, któremu odpowiada i-ty węzeł
$NAME[i] = i \qquad 1 \leq i \leq n$ — nazwa zbioru i-tego el.
$FATHER[i] = 0 \qquad 1 \leq i \leq n$ — czyli każde drzewo to pojedynczy węzeł.
$ROOT[i] = i \qquad 1 \leq i \leq n$ — id węzła-korzenia i-tego zbioru
na początku węzeł = korzeń

```
Find(i)
{
    opróżnij listę LIST;
    v=i;
    while (FATHER[v] ≠ ∅ )
    {
        dodaj v do LIST;
        v=FATHER[v];
    }
    { v jest teraz korzeniem }
    foreach (w ∈ LIST)
        FATHER[w]=v;
    return NAME[v];
}
```

```
Union(i,j,k)
{
    if (COUNT[ROOT[i]] > COUNT[ROOT[j]] )
        zamieniamy rolami i z j;
    LARGE=ROOT[j];
    SMALL=ROOT[i];
    FATHER[SMALL]=LARGE;
    COUNT[LARGE]=COUNT[LARGE]+COUNT[SMALL];
    NAME[LARGE]=k;
    ROOT[k]=LARGE;
}
```

Złożoność

n instrukcji Union-Find teraz ma złożoność: $O(n \log^* n)$.

$$log^* n = G(n) = \arg\min_i F(i) \geq n$$

$$\begin{aligned} F(0) &= 1, \\ F(i) &= 2^{F(i-1)} \qquad i > 0 \end{aligned}$$

i	$F(i)$
0	1
1	2
2	4
3	16
4	65536
5	2^{65536}

Algorytm Tarjana

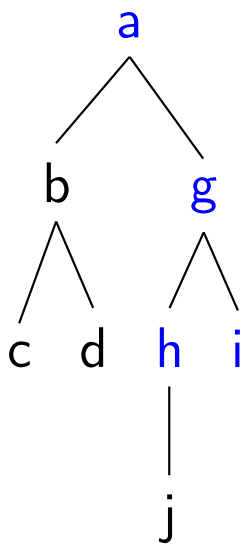
- Mamy drzewo T i chcemy wyznaczać pierwszego wspólnego przodka dla zadanych par wierzchołków drzewa. Czyli taki przodek p i q, który jest przodkiem obu wierzchołków i ma najwyższy stopień (najgłębszy).
- Algorytm offline: dla zadanych z góry par wierzchołków znaleźć odpowiedzi.

Oznaczenia:

- r — korzeń drzewa.
- $n.children$ — dzieci wierzchołka n.
- P zbiór par wierzchołków dla których szukamy wspólnych przodków.
- W pierwszym kroku wykonywane jest wywołanie:
  `Tarjan(r)`
- Będziemy korzystać z operacji na zbiorach rozłącznych:
  Find, Union, MakeSet

```
function Init (T)
    foreach v in T
        v.color = white;
end
```

```
function MakeSet(x)
    x.father = 0;
    x.count = 1;
end
```




```
function Tarjan(u)
  MakeSet(u)
  u.ancestor = u;
  foreach v in u.children do
    Tarjan(v);
    Union(u,v);
    Find(u).ancestor = u;
  u.color = black;
  foreach v such that ⟨u, v⟩ in P do
    if v.color == black then
      wynik += [(u,v) -> Find(v).ancestor];
end
```

Jak optymalnie dobrać strukturę P? Set of sets...

Drzewa przedziałowe

- Drzewo przechowuje pewne informacje o przedziałach a nie tylko punktach/wartościach
- Każdy wierzchołek opisany jest więc przez punkt i dodatkowe informacje. Np. szerokość przedziału, ale często i inne parametry jak liczba punktów, wartości średnie dużych przedziałów/poddrzew.

Czasy operacji na drzewie: $O(n \log n)$ (zakłada się, że mamy realizację przez drzewa binarne zrównoważone, np. czerwono–czarne)
Przydatne w przeróżnych zastosowaniach:

- Aktywne przechowywanie „zajętych" przedziałów.
- Czy punkt należy do któregoś z przedziałów?
- Wszelkie inne dynamiczne operacje na przedziałach, jak łączenie, usuwanie, ...
- Ile jest punktów/kluczy pomiędzy a i b?

Grafy – problem silnie spójnych składowych

- Silnie spójną składową grafu $G = (V, E)$ jest maksymalny zbiór $U \subseteq V$ taki, że dla każdej pary wierzchołków v i u istnieje ścieżka z v do u i z u do v.
- Część algorytmów grafowych wykorzystuje algorytm podziału na silnie spójne składowe. Następnie rozwiązuje w ramach składowych podproblemy i finalnie łączy rezultaty.
- Zobaczymy, że można to zrobić za pomocą dwukrotnego przeszukiwania w głąb.

Transpozycja grafu $G(V, E)$ to $G^T(V, E^T)$ gdzie $E^T = \{(u, v) : (v, u) \in E\}$.

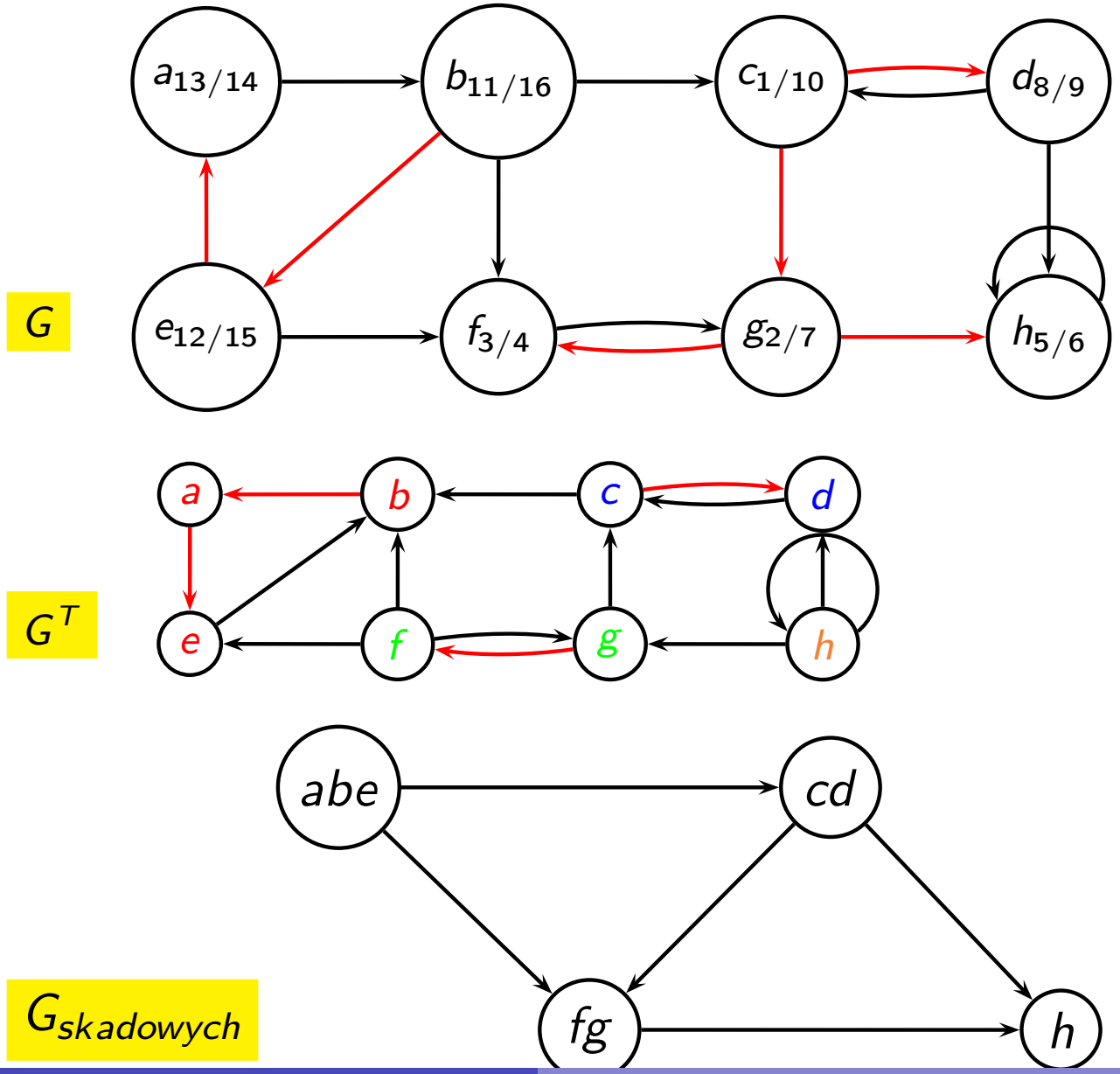
Ważne: Złożoność wyznaczenia G^T wynosi $O(V + E)$.

Własność: G i G^T składają się z tych samych silnie spójnych składowych. Jeśli u i v są osiągalne jeden z drugiego w G to i w G^T.

W celu wyznaczenia silnie spójnych składowych będziemy stosować dwa przeszukiwania w głąb, jedno dla grafu G a drugie dla grafu G^T.

```
procedure SilnieSpójneSkładowe(G(V, E))
    • wykonaj DFS(G) zapamiętując f[u] (czas
        przetworzenia każdego wierzchołka u);
    • wyznacz G^T;
    • wykonaj DFS(G^T(V, E^T)), przy czym główna pętla DFS
        analizuje wierzchołki w kolejności malejących
        wartości f[u];
    • wypisz wierzchołki z każdego drzewa lasu
            przeszukiwania w głąb dla G^T jako oddzielną
            składową;
end
```

[składowe = drzewa z czerwonych krawędzi]



- Warto zauważyć: jeśli u i v należą do tej samej silnie spójnej składowej (sss) i wierzchołek w leży na ścieżce z u do v lub z v do u, to
  w musi należeć do tej samej silnie spójnej składowej.

  Wiemy że istnieje $u \rightsquigarrow w$,
  mamy także: $w \rightsquigarrow v \rightsquigarrow u = w \rightsquigarrow u$.

- DFS zawsze umieści wierzchołki danej silnie spójnej składowej w tym samym drzewie przeszukiwania w głąb.

  Niech r będzie pierwszym odwiedzonym wierzchołkiem pewnej silnie spójnej składowej. Wtedy pozostałe wierzchołki tej SSS są białe.
  - Z właściwości DFS mamy, że każdy wierzchołek osiągalny z r i nieodwiedzony będzie odwiedzony i dopisany do drzewa do którego należy r.
  - Z drugiej strony wiemy, że istnieje droga z r do każdego wierzchołka silnie spójnej składowej. Czyli drzewo z r zbudowane przez DFS będzie zawierać wszystkie osiągalne z r wierzchołki.

- Praprzodek $\phi(u)$ wierzchołka u:
  $\phi(u) = w \ : \ u \rightsquigarrow w \wedge f[w]$ jest największe dla DFS(G)

  $f[w]$ – czas przetworzenia w przez DFS(G) (nie G^T!!).

  Właściwości praprzodka:

  - możliwe jest że: $\phi(u) = u$.
  - co daje: $f[u] \leq f[\phi(u)] \qquad (*)$
  - $u \rightsquigarrow v \Rightarrow f[\phi(u)] \geq f[\phi(v)]$
    ponieważ $\{w : v \rightsquigarrow w\} \subseteq \{w : u \rightsquigarrow w\}$.
  - Z powyższego: $f[\phi(u)] \geq f[\phi(\phi(u))]$, a z $(*)$ mamy
    $f[\phi(u)] \leq f[\phi(\phi(u))]$,
    czyli: $f[\phi(u)] = f[\phi(\phi(u))]$,
    a ostatecznie mamy $\phi(u) = \phi(\phi(u))$ (ponieważ nie ma 2 wierzchołków o tym samym czasie przetworzenia).

- W każdym przeszukiwaniu w głąb praprzodek $\phi(u)$ jest przodkiem u.
  - gdy $\phi(u) = u$ — oczywiste.
  - wpp. — rozważymy kolory $\phi(u)$ w kroku $d[u]$:
  - $1 - \phi(u)$ — niebieski (przetworzony)

  wtedy: $f[\phi(u)] < f[u]$ — sprzeczność z wł. praprzodka.
  - $2 - \phi(u)$ — czerwony (przetwarzany), to jest przodkiem (doszliśmy z $\phi(u)$ do u!).
  - $3 - \phi(u)$ — biały (nieodwiedzony)

  u.....białe.....$\phi(u)$ — $f[\phi(u)] < f[u]$ (czyli nie jest praprzodkiem)

  u.....niebiały(t)..białe.....$\phi(u)$ — t–czerwony (od nieb. do białego nie ma krawędzi). Wtedy $\phi(u)$ stanie się potomkiem t co daje: $f[\phi(u)] < f[t]$, czyli zły wybór praprzodka (sprzeczność).

- Z powyższego i z definicji praprzodka:
  u i $\phi(u)$ leżą w tej samej silnie spójnej składowej (istnieje ścieżka z u do $\phi(u)$ i odwrotnie).
- Dwa wierzchołki u i v leżą w tej samej silnie spójnej składowej $\iff$ mają tego samego praprzodka. (!!)

  $\Rightarrow$

  Gdy u i v w tej samej składowej, to każdy w osiągalny z u jest osiągalny z v i odwrotnie. Czyli $\phi(u) = \phi(v)$.

  $\Leftarrow$

  $\phi(u) = \phi(v)$. Wiemy też, że u i $\phi(u)$ należą do tej samej silnie spójnej składowej. To samo zachodzi dla v. Czyli u i v są w tej samej sss.
- Konkluzja: Praprzodek r + wierzchołki dla których r jest praprzodkiem tworzą sss!

- Przeszukiwanie DFS dla G^T rozpoczyna się od wierzchołka r z największym czasem przetworzenia $f[.]$. Czyli r musi być praprzodkiem.
  - Inne wierzchołki silnie spójnej składowej z r, to te dla których r jest praprzodkiem!
  - Czyli są to wszystkie wierzchołki, dla których r jest osiągalne i nie jest osiągalny żaden inny w. z $f[.]$ większym od $f[r]$ (czyli wszystkie $v \rightsquigarrow r$).
  - Czyli takie, które są osiągalne z r w G^T.

  - - Po wyznaczeniu pierwszego drzewa z DFS dla G^T (pierwszej sss). DFS jest kontynuowane i rozpoczyna od nieodwiedzonego jeszcze wierzchołka r' o maksymalnym $f[.]$.
  Jak poprzednio (nieodwiedzone) wierzchołki z których r' jest osiągalny składają się na sss. Czyli wierzchołki nieodwiedzone osiągalne z r' w G^T składają się na kolejną sss.

SSS, Punkty artykulacji i mosty

- SSS vs drogi miasta – wyznaczanie krytycznych odcinków dróg.
- Punkt artykulacji = wierzchołek (*skrzyżowanie*), którego usunięcie rozspójnia graf.
- Most = krawędź, której usunięcie rozspójnia graf.

Zadania

- Analiza algorytmu.
- Jak wyznaczyć graf składowych? Podaj algorytm.
- Jak wyznaczyć punkty artykulacji i mosty? Podaj algorytm.

Problem spełnialności reguł logicznych — SAT

$$L = x_1 \oplus_1 x_2 \oplus_2 \ldots \oplus_{n-1} x_n$$

- Problem SAT: Czy zadana reguła logiczna L oparta o zmienne logiczne $x_1, \ldots, x_n$ jest spełnialna?
- Czyli czy istnieje pewne przypisanie wartości prawda lub fałsz zmiennym x_i, dla którego reguła L była spełniona?
- W ogólności wyrażenie L może składać się z różnych operatorów logicznych: $\neg, \vee, \wedge, \Longrightarrow, XOR, \forall, \exists, \ldots$
- Dla większości typów reguł logicznych nie ma efektywnych (wielomianowych) algorytmów. Co pozostawia testowanie 2^n możliwych układów prawda/falsz.

Różne formy reprezentacji reguł logicznych. Np. CNF i DNF

- CNF (conjunction normal form)

$$L = (p_{11} \vee \ldots \vee p_{1k_1}) \wedge (p_{21} \vee \ldots \vee p_{2k_2}) \wedge \ldots (p_{n1} \vee \ldots \vee p_{nk_n})$$

- DNF (disjunction normal form)

$$L = (p_{11} \wedge \ldots \wedge p_{1k_1}) \vee (p_{21} \wedge \ldots \wedge p_{2k_2}) \vee \ldots (p_{n1} \wedge \ldots \wedge p_{nk_n})$$

gdzie p_{ij} to pewna zmienna x_q lub $\neg x_q$.

k-CNF i 2-CNF

- k-CNF (conjunction normal form)

$$L = (p_{11} \vee \ldots \vee p_{1k}) \wedge (p_{21} \vee \ldots \vee p_{2k}) \wedge \ldots (p_{n1} \vee \ldots \vee p_{nk})$$

- 2-CNF (conjunction normal form)

$$L = (p_{11} \vee p_{12}) \wedge (p_{21} \vee p_{22}) \wedge \ldots (p_{n1} \vee p_{n2})$$

Dla dowolnej postaci reguł logicznych problem jest NP-zupełny
Dla 3-CNF problem jest NP-zupełny
Dla 2-CNF problem ma efektywny algorytm!

Algorytm spełnialności reguł typu 2-CNF

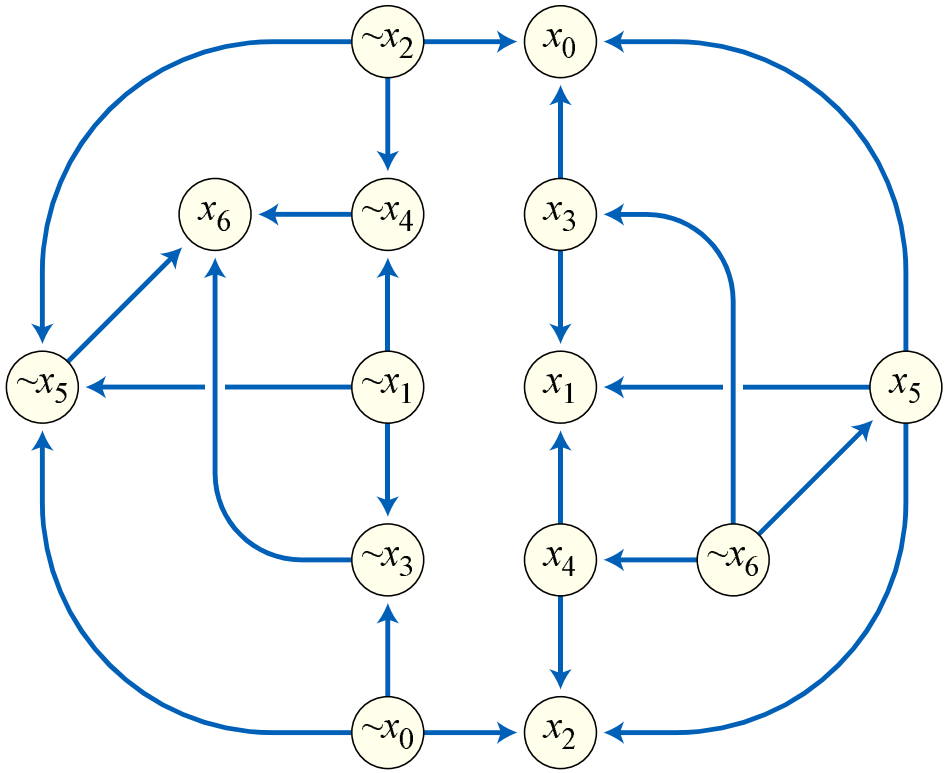
- Przejście z 2-CNF do postaci koniunkcji implikacji
- Analiza silnie spójnych składowych
- Ustalenie konsekwencji wynikających z SSS
- Możliwość wyznaczenie wartości zmiennych x_i realizujących spełnialność formuły, jeśli tylko to możliwe.

Przejście z 2-CNF do postaci koniunkcji implikacji

$$(x_0 \vee \neg x_3) \equiv (\neg x_0 \implies \neg x_3) \equiv (x_3 \implies x_0)$$

p	q	$p \implies q$	$\neg q \implies \neg p$	$\neg p \vee q$
0	0	1	1	1
0	1	1	1	1
1	0	0	0	0
1	1	1	1	1

$$(x_0 \vee x_2) \wedge (x_0 \vee \neg x_3) \wedge (x_1 \vee \neg x_3) \wedge (x_1 \vee \neg x_4) \wedge$$
$$(x_2 \vee \neg x_4) \wedge (x_0 \vee \neg x_5) \wedge (x_1 \vee \neg x_5) \wedge (x_2 \vee \neg x_5) \wedge$$
$$(x_3 \vee x_6) \wedge (x_4 \vee x_6) \wedge (x_5 \vee x_6)$$



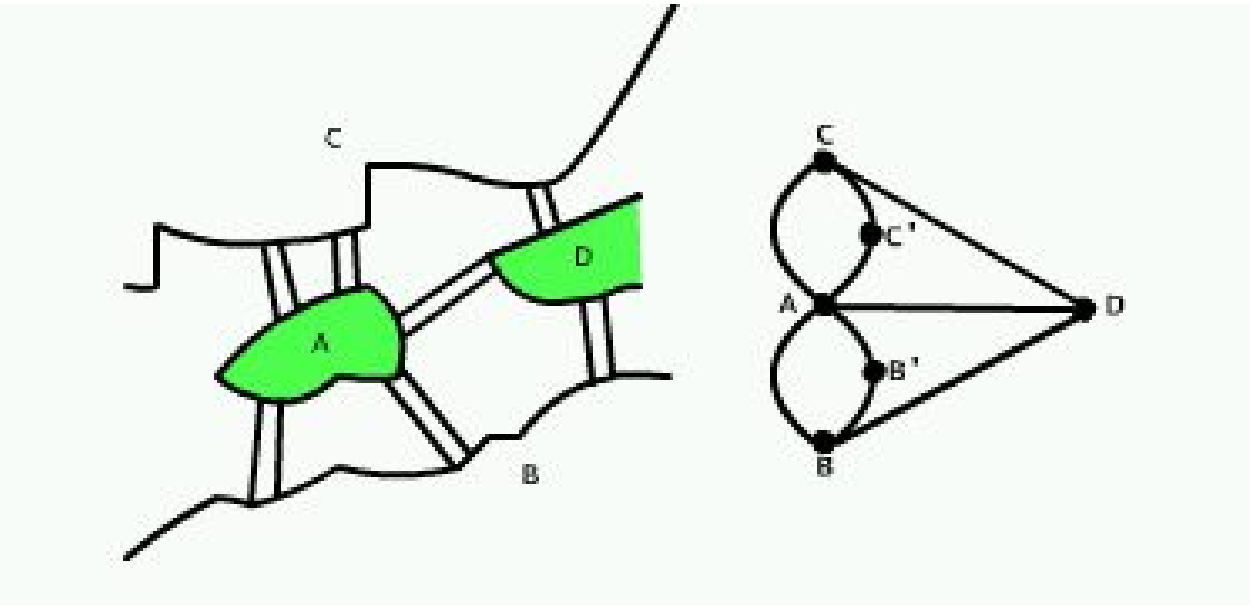
```
function Test_2CNF(CNF_rule)
    CNF_rule -> Graf implikacji G
    SSS = Silnie_spojne_skladowe(G)
    foreach v in V do
        if SSS[v] == SSS[¬v] then
            return false ;
    return true ;
end
```

Zmienne należące do jednej SSS składowej muszą mieć taką samą wartość!
Czyli nie mogą do SSS należeć: x_k i $\neg x_k$.

```
function Test_2CNF(CNF_rule)
    CNF_rule → Graf implikacji G
    SSS = Silnie_spójne_składowe(G)
    foreach v in V do
        if SSS[v] == SSS[¬v] then return false;
    Tworzenie grafu silnie spójnych składowych
    L = list of SSS; porządek zgodny z drugim DFS
    foreach c in L do
        if all v in c bez przypisanych wartości
            ustawić wszystkie v na false
            ustawić wartości komplementarne ¬v na true
        else
            nieustalone wartości v na wartość dowolnej
            ustalonej v' z tej składowej c
            (a komplementarne ¬v na ich negacje)
    return wektor wartości zmiennych x;
end
```

Tak wyznaczone wartości zmiennych logicznych spełniają regułę. Nieustawione wartości x_i mogą mieć dowolne wartości.

Cykl Eulera



Cyklem Eulera nazywamy taki cykl w grafie $G(V, E)$ (skierowany bądź nie), który przechodzi przez wszystkie krawędzie dokładnie raz i przez każdy wierzchołek przynajmniej raz.

Dla grafu spójnego skierowanego:

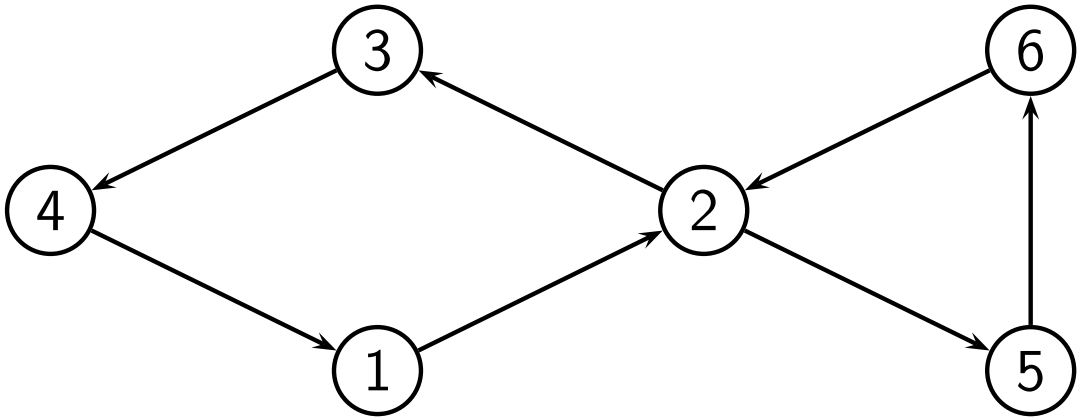
Cykl Eulera istnieje wtedy i tylko wtedy gdy dla każdego wierzchołka liczba krawędzi wchodzących i wychodzących jest równa.

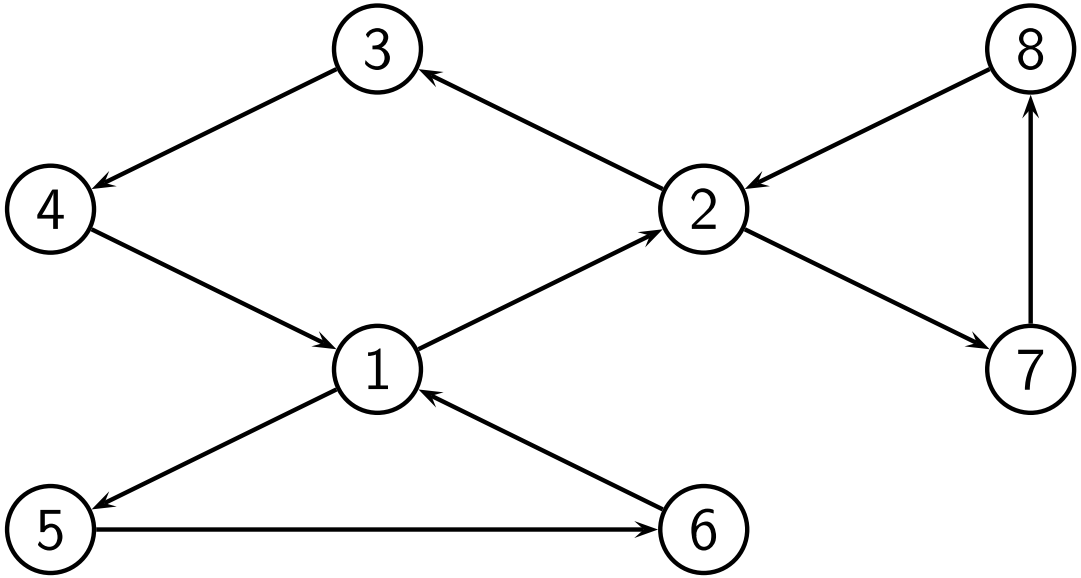
Dla grafu spójnego nieskierowanego:

Cykl Eulera istnieje wtedy i tylko wtedy gdy stopień każdego wierzchołka jest parzysty.

```
CyklEulera(G, u) {
    push(STOS,u);
    while (STOS != ∅){
        v=top(STOS);
        if (S[v]==∅)
        {
            pop(STOS);
            LISTA ← v;    {wstaw v na początek}
        }else
        {
            w=pop(S[v]);
            push(STOS,w);
        }
    }
    return LISTA;
}
```

1–2–5–6–2–3–4–1

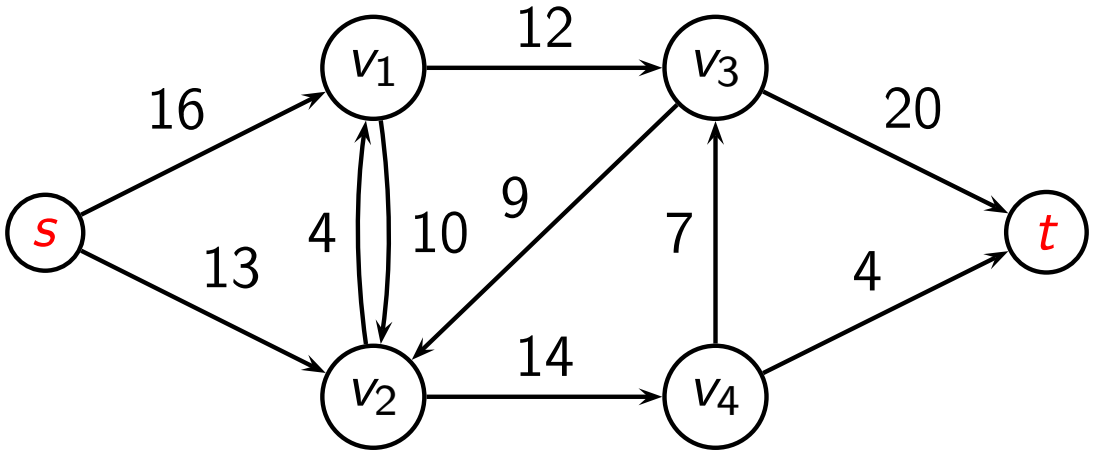




1–2–7–8–2–3–4–1–5–6–1

Problem maksymalnego przepływu

- Sieci przepływowe
- Źródło (s) i ujście (t)
- Transport (przepływ) towaru od źródła wytwarzania do ujścia (miejsca zużycia).



- Zastosowania:
  - Modelowanie przepływu cieczy w sieci kanałów, przepustowości dróg publicznych.
  - Przepływ informacji w sieciach komunikacyjnych (Internet, telefonia cyfrowa).
  - Modelowanie przepływu prądu w sieciach energetycznych.

- Wierzchołki – miejscowości (czy komputery, routery, switche, ...)
- Krawędzie – kanały przepływu (informacji, cieczy, towaru, ...)
- Waga krawędzi definiuje maksymalną przepustowość krawędzi (łącza).
- Wierzchołki na ścieżkach pomiędzy źródłem i ujściem są wykorzystywane do realizacji przepływu.
- Szybkość wpływania do wierzchołka musi być taka sama jak wypływania (własność zachowania przepływu)!
- Towar/Informacja nie jest przetrzymywany(-a) w wierzchołkach.
- Maksymalny przepływ = maksymalna prędkość z jaką można transportować towar ze źródła do ujścia.

- Sieć przepływowa to graf $G(V, E)$, gdzie dla każdej krawędzi (u, v) mamy określone $c(u, v) \geq 0$, co odpowiada przepustowości krawędzi. Jeśli $(u, v) \notin E$ to $c(u, v) = 0$.
- Sieć przepływowa ma wyróżnione dwa wierzchołki: źródło s i ujście t.

Przepływ sieci

Przepływ sieci G to funkcja $f : V \times V \to \Re$, która spełnia trzy warunki:

- Warunek przepustowości: dla wszystkich u i v mamy

$$f(u, v) \leq c(u, v).$$

  - Oznacza to, że przepływ nie może przekraczać przepustowości.
- Warunek skośnej symetryczności: dla wszystkich u i v mamy

$$f(u, v) = -f(v, u).$$

  - Mamy więc: $f(u, u) = -f(u, u)$, czyli $f(u, u) = 0$.

- Warunek zachowania przepływu: dla każdego $u \in V - \{s, t\}$ mamy

$$\sum_{v \in V} f(u, v) = 0.$$

  - Warunek wymusza aby cały towar wpływający do u wypływał z u

- Z warunku skośnej symetryczności i zachowania przepływu mamy:

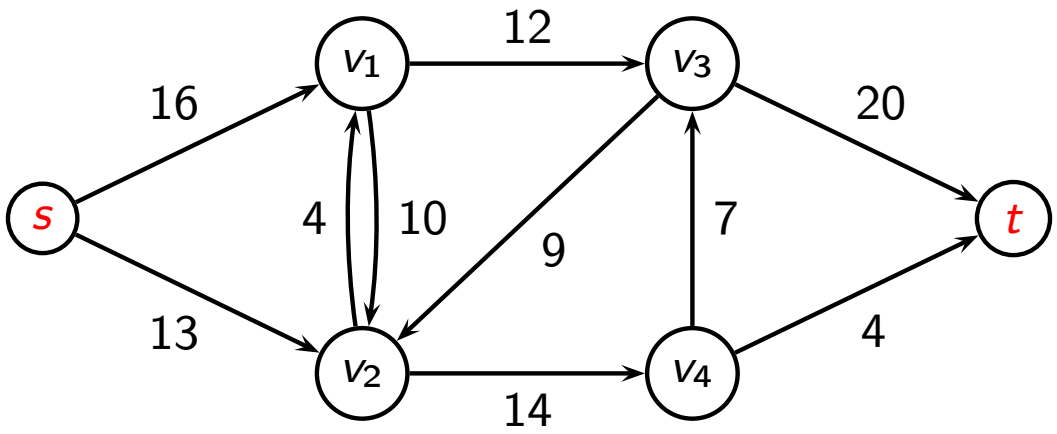
Dla każdego $v \in V - \{s, t\}$

$$\sum_{u \in V} f(u, v) = 0.$$

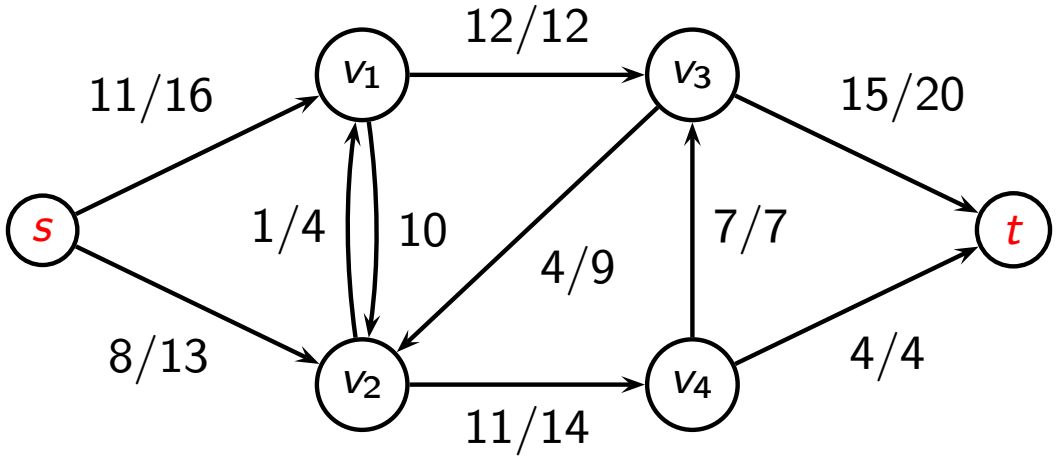
- $f(u, v)$ nazywa się przepływem netto z wierzchołka u do v (może być dodatni bądź ujemny).
- Wartość przepływu $|f|$ definiuje się jako

$$|f| = \sum_{v \in V} f(s, v)$$

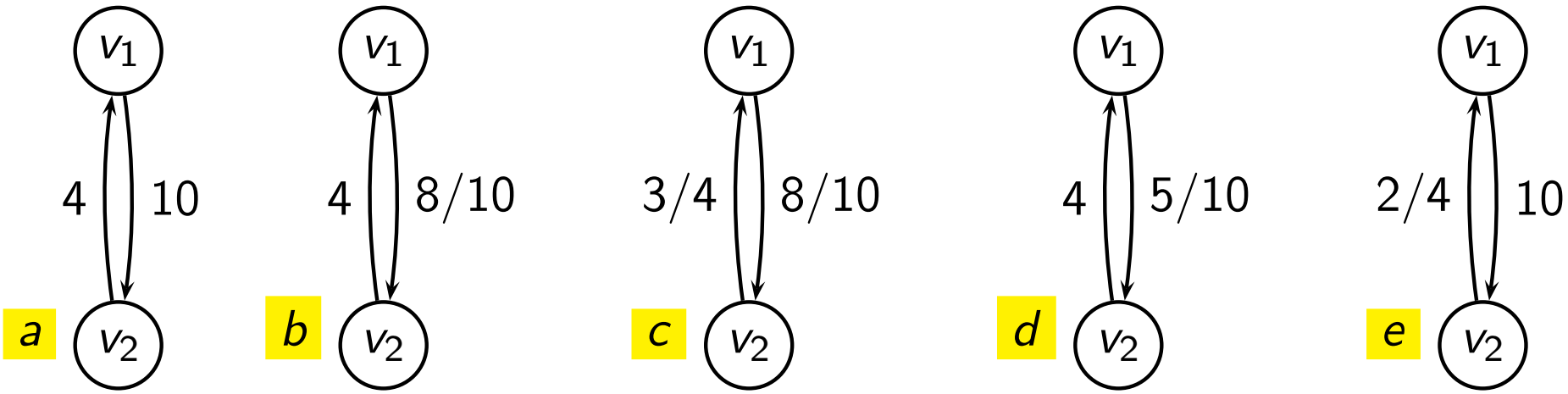
łączny przepływ netto wypływający ze źródła s. Jak potem zobaczymy będzie równy całkowitemu wpływowi do t.



przepływ / przepustowość



UWAGA: Prezentujemy TYLKO dodatni przepływ netto ($f(u, v) > 0$).



a – przepustowości G

b – 8 z v_1 do v_2

c – 8 z v_1 do v_2 i 3 z v_2 do v_1

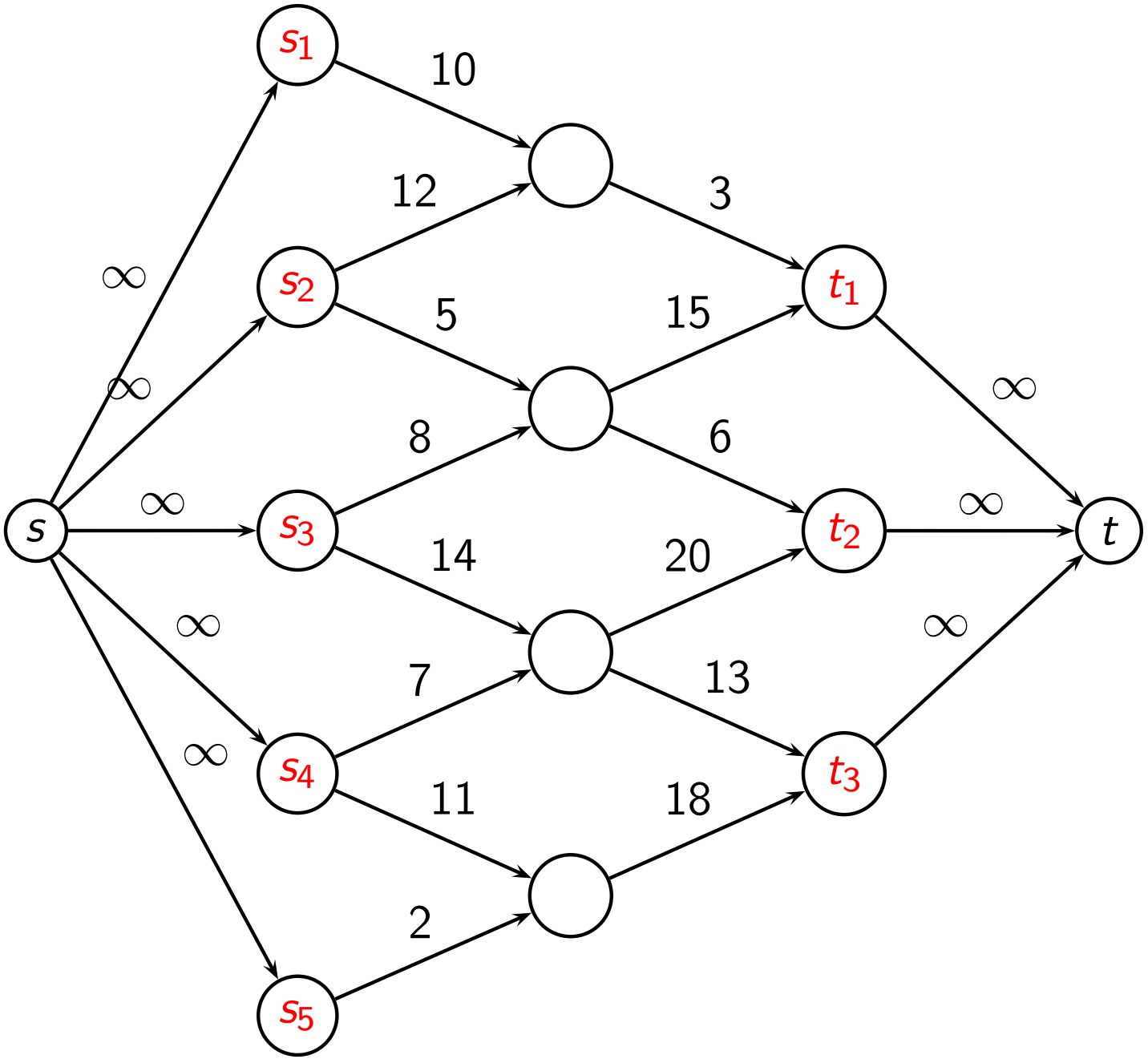
d – po REDUKCJI: 5 z v_1 do v_2

- Czyli nie ma sensu wożenie tego samego towaru z v_1 do v_2 i z v_2 do v_1.

e – dodatkowo 7 z v_2 do v_1

Sieci z wieloma źródłami i wieloma ujściami

- Problem maksymalnego przepływu można także zdefiniować dla więcej niż jednego źródła i większej liczby ujść.
- Na szczęście problem maksymalnego przepływu w takim przypadku jest tak samo trudny. Co więcej problemy są równoważne.
- Można wprowadzić super-źródła s które będzie połączone z każdym źródłem s_i $(i = 1, \ldots, n)$ krawędzią o przepustowości ∞. Wprowadzamy analogicznie super-ujście t, do którego wpadają krawędzie z każdego ujścia t_j $(j = 1, \ldots, m)$ także o nieskończonej przepustowości.


s
s1
s2
s3
s4
s5
t1
t2
t3
t
∞
10
12
3
5
15
8
6
14
20
7
13
11
18
2

- Zmiana notacji:

$$V - s \equiv V - \{s\}$$

- Niech

$$f(X, Y) = \sum_{x \in X} \sum_{y \in Y} f(x, y)$$

Wtedy prawo zachowania przepływu to: $f(u, V) = 0$ dla $u \in V - s - t$.

Lemat
Zachodzą następujące własności:

$$f(X, X) = 0 \qquad X \subseteq V$$

$$f(X, Y) = -f(Y, X) \qquad X, Y \subseteq V$$

Niech $X, Y, Z \subseteq V$ i $X \cap Y = \emptyset$, wtedy:

$$f(X \cup Y, Z) = f(X, Z) + f(Y, Z)$$

$$f(Z, X \cup Y) = f(Z, X) + f(Z, Y)$$

Możemy pokazać, że

$$|f| = f(V, t)$$

Czyli, że łączny przepływ jest równy całkowitemu wpływowi do ujścia.

$$\begin{aligned} |f| &= f(s, V) \\ &= f(V, V) - f(V - s, V) \\ &= f(V, V - s) \\ &= f(V, t) + f(V, V - s - t) \\ &= f(V, t) \end{aligned}$$

Metoda Forda-Fulkersona

- Start: $\forall_{u,v}\ f(u,v) = 0$
- Ścieżka powiększająca — ścieżka ze źródła s w G do ujścia t po której możemy przesłać większy przepływ.
- Iteracyjnie, przepływ jest powiększany o możliwości jakie daje każda ścieżka powiększająca.
- Koniec gdy nie istnieje żadna ścieżka powiększająca z s do t.

```
procedure Ford−Fulkerson
begin
    inicjowanie f na 0
    while istnieje ścieżka powiększająca p do
        powiększ przepływ f wzdłuż p
    return f
end
```

Jak wyznaczać ścieżki powiększające? Sieci residualne

- Sieć residualna dla grafu G składa się z krawędzi, które dopuszczają większy przepływ netto.
- Przepustowością residualną dla (u, v) nazywamy dodatkowy przepływ w G zdefiniowany poprzez:

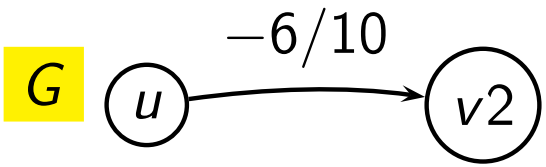
$$c_f(u, v) = c(u, v) - f(u, v).$$

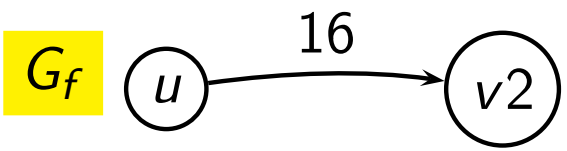
- np. $c(u, v) = 10$, $f(u, v) = 6$, to $c_f(u, v) = 4$

- Czyli przepustowość residualna to niewykorzystana przepustowość krawędzi w sieci grafu G.

- Gdy przepływ netto jest ujemny przepustowość residualna może być większa niż $c(u, v)$.

np. $c(u, v) = 10$, $f(u, v) = -6$, to $c_f(u, v) = 16$



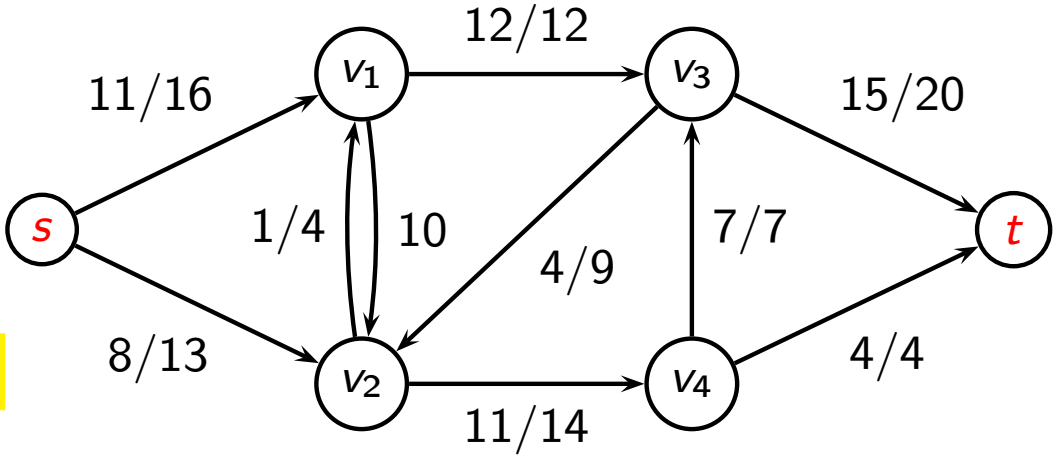


- Siecią residualną dla $G = (V, E)$ indukowaną przez f nazywamy sieć $G_f = (V, E_f)$ taką, że

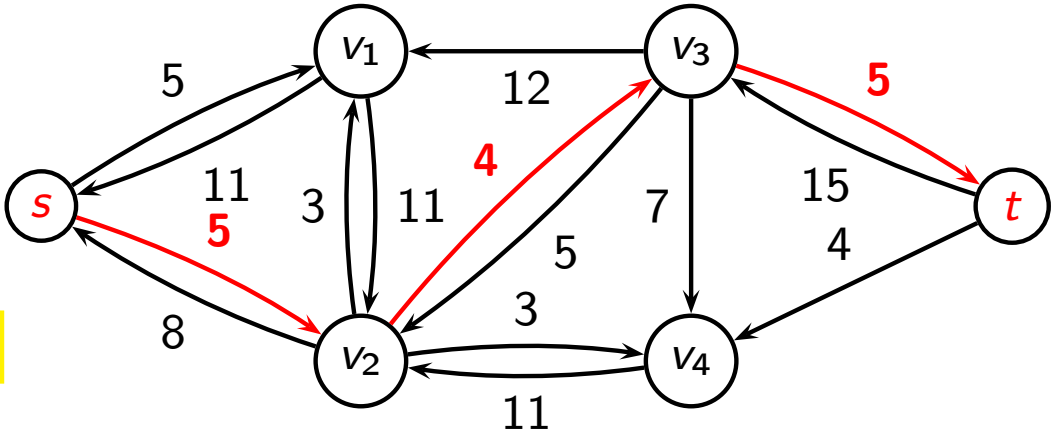
$$E_f = \{(u, v) \in V \times V : c_f(u, v) > 0\}.$$

Czyli sieć składa się z tych samych wierzchołków co G, ale krawędzie tworzą *przepustowości residualne* czyli krawędzie umożliwiające uzyskanie dodatniego przepływu netto.

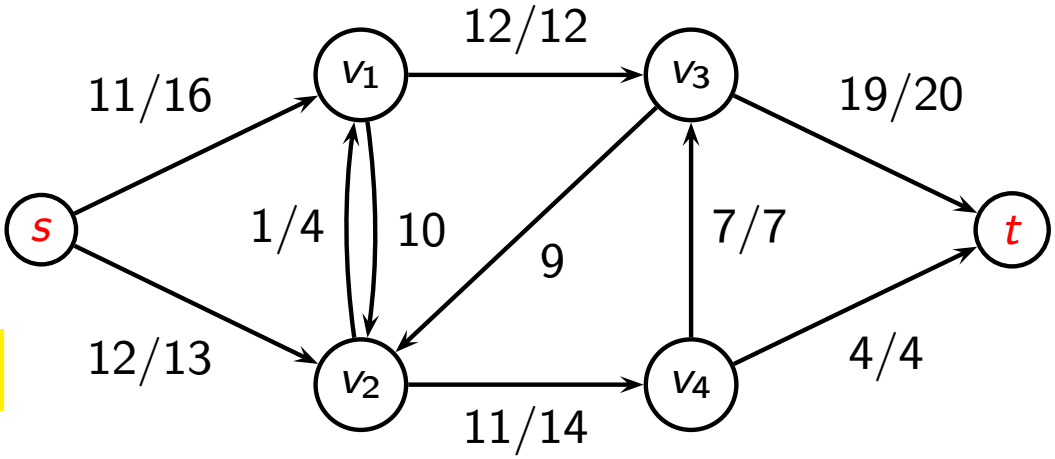
G, f


s
v1
v2
v3
v4
t
11/16
8/13
12/12
1/4
10
4/9
7/7
11/14
15/20
4/4

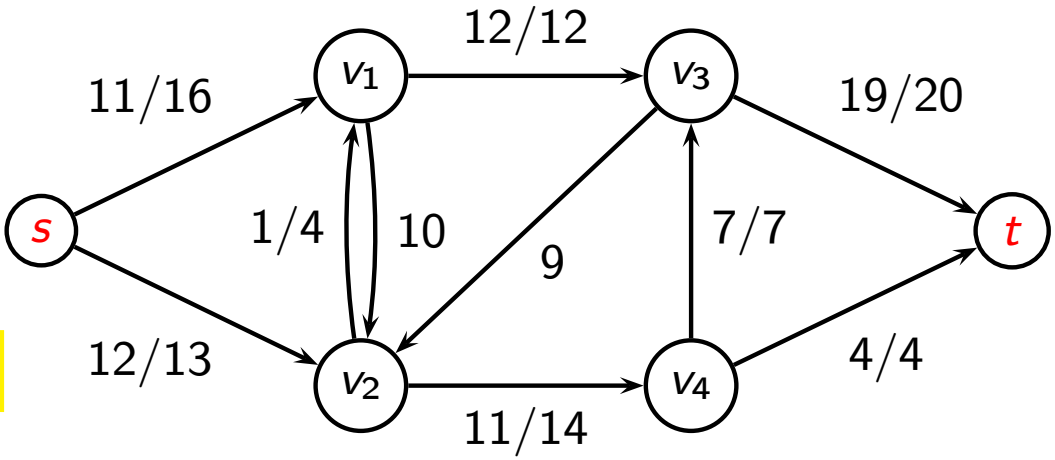
G_f


s
v1
v2
v3
v4
t
5
11
5
8
3
11
12
4
5
3
7
11
5
15
4

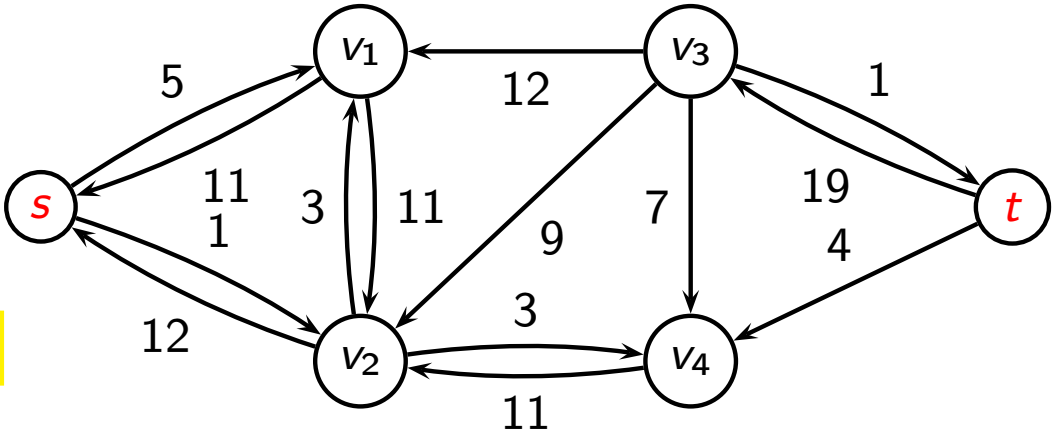
G, f'


s
v1
v2
v3
v4
t
11/16
12/13
12/12
1/4
10
9
7/7
11/14
19/20
4/4

G, f'


s
v1
v2
v3
v4
t
11/16
12/12
19/20
1/4
10
9
7/7
12/13
11/14
4/4

$G_{f'}$


s
v1
v2
v3
v4
t
5
12
1
11
1
3
11
9
7
19
4
12
3
11

- Liczba krawędzi residualnych $|E_f|$ jest ograniczona z góry przez $2|E|$.

Lemat

Mamy sieć G, źródło s, ujście t, przepływ f dla G, sieć residualną G_f dla sieci G indukowaną przez f. Niech f' oznacza przepływ w sieci G_f wtedy suma przepływów $f + f'$ jest przepływem w sieci G o wartości $|f + f'| = |f| + |f'|$

Dowód:

1. skośna symetryczność:

$$
\begin{aligned}
(f + f')(u, v) &= f(u, v) + f'(u, v) \\
&= -f(v, u) - f'(v, u) \\
&= -(f(v, u) + f'(v, u)) \\
&= -(f + f')(v, u)
\end{aligned}
$$

2. przepustowość

wiemy: $f'(u,v) \leq c_f(u,v)$

$$\begin{aligned}(f+f')(u,v) &= f(u,v) + f'(u,v) \\ &\leq f(u,v) + (c(u,v) - f(u,v)) \\ &= c(u,v)\end{aligned}$$

3. zachowanie przepływu
$\forall u \in V - s - t$

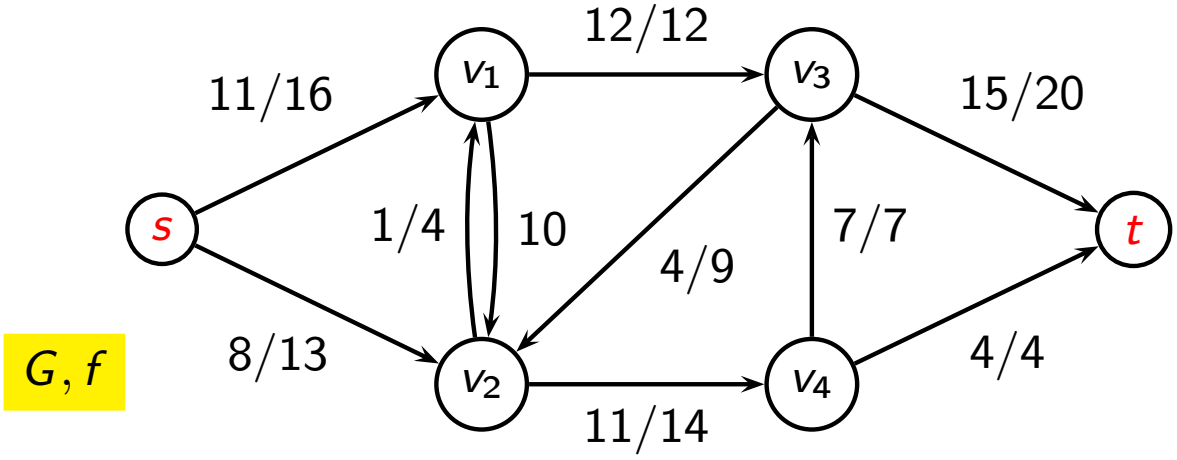
$$\begin{aligned}
\sum_{v \in V} (f + f')(u, v) &= \sum_{v \in V} (f(u, v) + f'(u, v)) \\
&= \sum_{v \in V} f(u, v) + \sum_{v \in V} f'(u, v) \\
&= 0 + 0
\end{aligned}$$

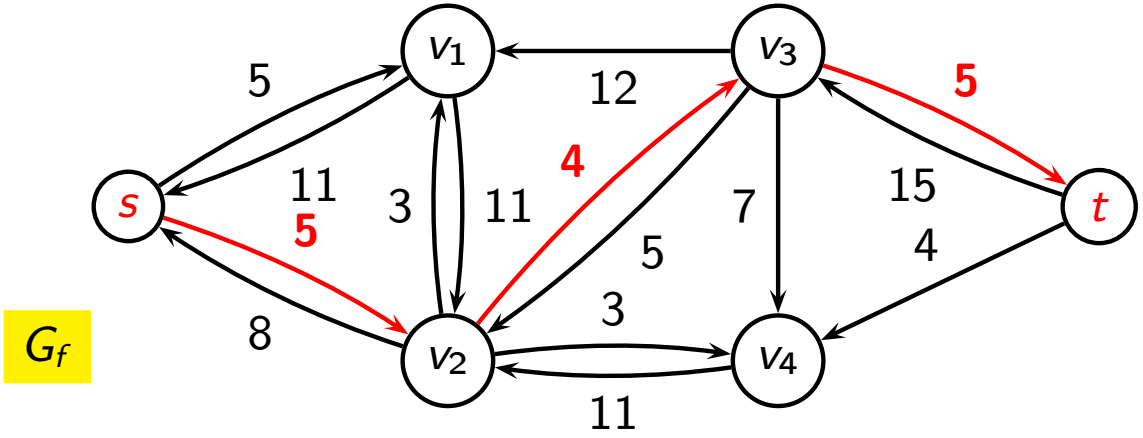
Wartość przepływu $|f + f'|$ wynosi:

$$
\begin{aligned}
|f + f'| &= \sum_{v \in V} (f + f')(s, v) \\
&= \sum_{v \in V} (f(s, v) + f'(s, v)) \\
&= \sum_{v \in V} f(s, v) + \sum_{v \in V} f'(s, v) \\
&= |f| + |f'|
\end{aligned}
$$

Ścieżki powiększające – formalnie

Dla sieci G i przepływu f ścieżką powiększającą p jest każda ścieżka z źródła s do ujścia t w sieci residualnej G_f.





Przepustowość residualna ścieżki p:

$$c_f(p) = \min\{c_f(u, v) : (u, v) \text{ leży na ścieżce } p\}$$

Lemat

Mamy sieć G, przepływ f sieci G, ścieżkę powiększającą p w sieci G_f. Zdefiniujmy funkcję $f_p : V \times V \to \Re$:

$$f_p(u,v) = \begin{cases} c_f(p) & \text{jeśli } (u,v) \text{ jest na } p \\ -c_f(p) & \text{jeśli } (v,u) \text{ jest na } p \\ 0 & wp.p. \end{cases}$$

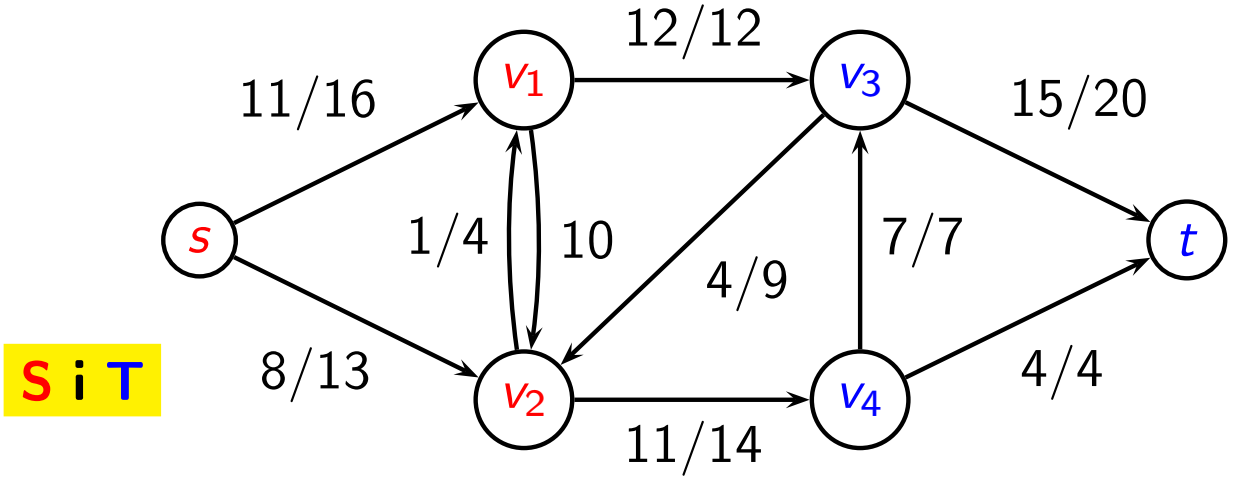
Wtedy f_p jest przepływem w sieci G_f o wartości $|f_p| = c_f(p) > 0$.

- Z powyższego lematu wynika, że dodając przepływ f_p do f otrzymujemy większy przepływ w G.

Przekroje w sieciach

Mamy sieć $G = (V, E)$. Przekrojem (S, T) nazywamy taki podział V na S i T, że $s \in S$ i $t \in T$, i $T = V - S$.

- Wtedy przepływ netto przez przekrój to $f(S, T)$.
- A przepustowość przekroju: $c(S, T)$.



Niech f będzie przepływem netto w G, a (S, T) przekrojem. Wtedy przepływ przekroju równy jest przepływowi sieci f(S,T)=|f|.
Dowód:

$$\begin{aligned} f(S,T) &= f(S,V) - f(S,S) \\ &= f(S,V) \\ &= f(s,V) + f(S-s,V) \\ &= f(s,V) \\ &= |f| \end{aligned}$$

Skąd wnioskujemy:

Wartość dowolnego przepływu f w G jest nie większa niż przepustowość dowolnego przekroju w G.

Dowód:

$$\begin{aligned} |f| &= f(S,T) \\ &= \sum_{u \in S} \sum_{v \in T} f(u,v) \\ &\leq \sum_{u \in S} \sum_{v \in T} c(u,v) \\ &= c(S,T) \end{aligned}$$

Twierdzenie o maksymalnym przepływie i minimalnym przekroju

Dla sieci G o źródle s i ujściu t następujące trzy warunki są równoważne:

- Przepływ f jest maksymalny.
- Sieć residualna G_f nie zawiera ścieżek powiększających.
- Dla pewnego przekroju (S, T) w G zachodzi: $|f| = c(S, T)$.

$1 \Rightarrow 2$

załóżmy przeciwnie: f przepływ maksymalny, a G_f ma ścieżkę powiększającą. Wtedy jednak wiemy, że $f + f_p$ będzie przepływem większym niż f.

$2 \Rightarrow 3$

Niech S będzie zbiorem wierzchołków osiągalnych z s:

$$S = \{v \in V : s \rightsquigarrow v \text{ w } G_f\}.$$

$T = V - S$.

Ponieważ nie istnieje ścieżka powiększająca z s do t w G_f, to mamy $f(u, v) = c(u, v)$, gdy tylko $u \in S$ i $v \in T$. W przeciwnym przypadku v byłby w S. Mamy więc: $f(S, T) = c(S, T) = |f|$.

$3 \Rightarrow 1$

dla każdego przekroju zachodzi: $|f| \leq c(S, T)$. Czyli $|f| = c(S, T)$ oznacza, że jest to maksymalny pośród możliwych przepływów.

```
procedure Ford−Fulkerson(G,s,t)
begin
    foreach (u, v) ∈ E do
        begin  f(u,v)=0; f(v,u)=0; end

    while istnieje ścieżka p z s do t w G_f (graf
        residualny ) do
    begin
        c_f(p) = min{c_f(u, v) : (u, v) jest na p}
        for każda krawędź (u, v) na p do
            begin f(u,v) = f(u,v) + c_f(p);
                f(v,u) = −f(u,v)
            end
    end
end
```

- Gdy założyć, że przepustowość będzie wyrażana w liczbach całkowitych to złożoność algorytmu wynosi $O(E|f^*|)$, gdzie f^* jest maksymalnym przepływem.
Wynika to z faktu, iż znalezienie ścieżki powiększającej w G_f jest proporcjonalne do E (np. poprzez przeszukiwanie w głąb), a każde powiększenie ścieżki może w najgorszym razie poprawić przepływ netto o 1. Warto używać, gdy $|f^*|$ jest nie za duże.
- Zastępując przeszukiwanie w głąb na przeszukiwanie wszerz aby znaleźć ścieżkę powiększającą w grafie residualnym złożoność redukuje się do $O(VE^2)$

Metody przedprzepływowe

- Aby wyobrazić sobie jak działa ta metoda lepiej myśleć o przepływie wody, która może płynąć tylko w dół.
- Na potrzeby algorytmu każdy wierzchołek będzie miał zbiornik o nieograniczonej pojemności. Każdy wierzchołek wraz z przyłączami rur umieszczamy na specjalnej platformie, której pozycja będzie podnoszona wraz z działaniem algorytmu.
- Wysokość wierzchołka będzie w istotny sposób wpływać na przepływ!!!
- Metoda przedprzepływowa nie będzie utrzymywała własności zachowania przepływu. Natomiast będzie spełniony warunek słabszy: $f(V, u) \geq 0$, który określa sumaryczny wpływ do pewnego wierzchołka u $(u \in V - s)$.

- u będzie wierzchołkiem nadmiarowym, gdy $f(V, u) > 0$ (nadmiar: $e[u] = f(V, u)$).
- Przepływ jest realizowany tylko do wierzchołka położonego niżej, ale funkcja przepływu może mieć wartości dodatnie skierowane do wierzchołka położonego powyżej.
- Wysokość źródła wynosi $|V|$, a ujścia 0 — to nie podlega zmianie.
- Wysokości pozostałych wierzchołków są ustawione na początkową wartość 0 i będą rosły.
- Startujemy z realizacją całego przepływu, wypływającego ze źródła wypełniając maksymalnie istniejące odprowadzające rury.
- Woda najpierw trafia do zbiornika a następnie jest kierowana do położonych niżej wierzchołków.
- Jednak problem polega na tym, że aby woda płynęła dalej wierzchołek musi być powyżej niewypełnionych jeszcze innych wierzchołków połączonych rurami. Tak może nie być.

- W takich przypadkach będziemy podnosić wysokość wierzchołka, aby nadmiar wody mógł spłynąć niżej.
- Wysokość takiego wierzchołka podnosimy o 1 więcej niż wysokość najniższego wierzchołka do którego prowadzi niewypełniona w całości rura.
  Dzięki temu pewien nadmiar wody będzie mógł spłynąć w dół.
- Po powtórzeniach tej operacji w końcu cała woda spłynie do ujścia. Jednak nie więcej niż to możliwe – dzięki ograniczeniu przepustowości krawędzi.

- Jednakże tak uzyskany przepływ może być *nielegalny* ponieważ może pozostać woda w zbiornikach wierzchołków, która nie ma jak spłynąć.
- Taki nadmiar cofany jest do źródła poprzez podnoszenie wysokości wierzchołków powyżej wysokości źródła $|V|$. A w algorytmie będzie to realizowane przez kasowanie przepływów nadmiarowych.
- Po opróżnieniu zbiorników uzyskujemy legalny i maksymalny przepływ.

Operacje PUSH i LIFT

- Algorytm składa się z szeregu wykonań operacji przesyłania nadmiaru wody (PUSH) i podnoszenia wysokości wierzchołków (LIFT).
- Funkcja $h : V \to N$ jest wysokością w sieci $G = (V, E)$ ze źródłem s i ujściem t, gdy:
  - $h(s) = |V|$,
  - $h(t) = 0$,
  - $h(u) \leq h(v) + 1$, gdy (u, v) jest krawędzią residualną – $(u, v) \in E_f$.

  Czyli gdy $h(u) > h(v) + 1$, to (u, v) nie jest krawędzią w grafie residualnym.

- Operację $PUSH(u, v)$ można stosować $\iff$ u jest wierzchołkiem nadmiarowym ($e[u] > 0$) i $h(u) = h(v) + 1$ (tj. jest coś do przelania i u jest bezpośrednio nad v).
- $e[u]$ będzie nadmiarem wody przechowywanym w wierzchołku u.
- $d_f(u, v)$ przechowuje informację o wielkości przepływu jaki można przesłać z u do v.

```
procedure PUSH(u,v)
    d_f(u, v) = min(e[u], c_f(u, v))
    f[u, v] = f[u, v] + d_f(u, v)
    f[v, u] = −f[u, v]
    e[u] = e[u] − d_f(u, v)
    e[v] = e[v] + d_f(u, v)
end
```

- Warto zwrócić uwagę, że gdy f jest przedprzepływem to po wykonaniu $PUSH(u, v)$ też będzie przedprzepływem.
- Operacja $PUSH(u, v)$ przesyłania jest nasycająca, gdy po jej wykonaniu $c_f(u, v) = 0$.
- Operacja $LIFT(u)$ jest wykonywana gdy:
  1. u jest nadmiarowy
  2. $\forall_v \ (u, v) \in E_f \Rightarrow h[u] \leq h[v]$ – tj. woda nie może się przelać.

```
procedure LIFT(u)
    h[u] = 1 + min(h[v] : (u, v) ∈ E_f)
end
```

- Operacja LIFT jest stosowana tylko, gdy woda mogła by się przelać z wierzchołka u, ale nie może.

```
procedure  Initialize_PreFlow (G,s)
    foreach u ∈ V do
        h[u]=0, e[u]=0;
    foreach (u, v) ∈ E do
        f[u,v]=0,  f[v,u]=0;
    h[s]=|V|;
    foreach u ∈ S[s] do
    begin
        f[s,u]=c[s,u];
        f[u,s]=−c[u,s];
        e[u]=c[s,u];
    end
end
```

```
procedure Generic_PreFlow_PUSH(G)
    Initialize_PreFlow (G,s);
    while można stosować operację przesłania
            lub podniesienia do
        wybierz jedną z operacji i wykonaj;
end
```

Złożoność metody przedprzepływowej

Ograniczenie na liczbę podnoszeń: $< 2|V|^2$.
Ograniczenie na liczbę przesłań nasycających: $< 2|V||E|$
Ograniczenie na liczbę przesłań nienasycających: $< 4|V|^2(|V| + |E|)$
Ostatecznie złożoność wynosi: $O(V^2E)$

Od $O(V^2E)$ do $O(V^3)$ — Podnieś i przesuń na początek

- Aby osiągnąć powyższą złożoność nowy algorytm będzie bazował na liście wierzchołków L, co będzie regulowało ściślej kolejnością wykonywanych operacji PUSH i LIFT.
- Lista ta będzie przeglądana od początku w poszukiwaniu wierzchołka nadmiarowego.
- W następnym kroku następuje całkowite rozładowanie takiego wierzchołka (poprzez wykonywanie operacji PUSH i LIFT aż do skutku).
- Jeśli wierzchołek podczas rozładowywania uległ podniesieniu zostaje przeniesiony na początek listy L, a główna pętla algorytmu rozpoczyna pracę od początku listy L (w poniższej implementacji tak naprawdę od drugiego wierzchołka).

```
Discharge(u) {
    while (e[u]>0)
    {
        v=currentS[u];
        if (v==NULL) // po ostatnim sąsiedzie
        {
            LIFT(u);
            currentS[u]=head[S[u]];
        }
        else if (c_f(u,v) > 0 && h[u]==h[v]+1 )
            PUSH(u,v);
        else currentS[u]=nextS[v];
    }
}
```

```
Lift_To_Front(G,s,t) {
    Initialize_PreFlow (G,s);
    L=V−s−t;
    foreach (v ∈ V − s − t )
        currentS[u]=head[S[u]];
    u=head[L];
    while (u != NULL )
    {
        oldHeight[u]=h[u];
        Discharge(u);
        if (h[u]>oldHeight )
            przesuń u na początek L;
        u=nextS[u];
    }
}
```

Algorytmy plecakowe

- Trzy różne problemy „ładowania" plecaka = trzy różne algorytmy
- Niewielka zmiana definicji problemu = wielka zmiana koncepcji optymalnego rozwiązania, wielka różnica w złożoności

Ciągły problem plecakowy

- Mamy plecak o pojemności W.
- Mamy dany zbiór N towarów. Każdy towar ma swoją cenę c_i i wielkość w_i.
- Każdego towaru można wziąć dowolną ilość lecz nie więcej niż w_i.
- Łącznie nie można wziąć więcej towarów niż W.
- Wartość załadowanego plecaka ma być największa.

Co rozwiązuje problem?

- Wyznaczmy względną wartość towarów:

$$q_i = \frac{c_i}{w_i}$$

- Teraz wiemy co jest najbardziej wartościowym towarem, czy to złoto, czy biała herbata, czy diamenty???
- Sortujemy ciąg q_i od najwartościowszego towaru.
- Bierzemy kolejno towary, zgodnie z posortowaniem, które mieszczą się w plecaku aż do pełnego wypełnienia (lub braku towarów).

```
PlecakCiagly(W, N, w, c) {
    for (i=1 to N )
        q_i = c_i/w_i
    sortowanie tablic w, c i q względem q (od największego)
    x=0;
    i=1;
    while (x < W && i <= N ){
        // weź tyle towaru i do plecaka ile wejdzie;
        x += min(w_i, W−x)
        i++;
    }
}
```

Złożoność: $O(N \log N)$ z powodu sortowania.

Dyskretny problem plecakowy

- Mamy plecak o pojemności W.
- Mamy dany zbiór N towarów. Każdy towar ma swoją cenę c_i i wielkość w_i.
- Każdy towar w_i można wziąć lub nie wziąć.
- Suma wielkości załadowanych towarów nie może przekraczać W.
- Wartość załadowanego plecaka ma być możliwie największa.

Co rozwiązuje problem?
Pełny przegląd możliwości: $O(2^N)$
Problem NP zupełny!

Dynamiczne rozwiązanie problemu dyskretnego

- Rozwiązanie pseudo-wielomianowe jest możliwe dzięki pewnym sprytnym założeniom...
- Załóżmy, że W jest wartością całkowitą a także każde z wartości w_i jest całkowite.
- Wtedy możemy inaczej spojrzeć na całość problemu:
  - Co zrobić możemy gdy mamy $W' = 1$ i jeden towar?
  - Co zrobić możemy gdy mamy $W' = 2$ i jeden towar?
  - Co zrobić możemy gdy mamy $W' = 3$ i jeden towar?

  - Co zrobić możemy gdy mamy $W' = 1$ i dwa towary?
  - Co zrobić możemy gdy mamy $W' = 2$ i dwa towary?
  - Co zrobić możemy gdy mamy $W' = 3$ i dwa towary?

Wariant I:

Załóżmy, że każdego towaru można wziąć $c \in N$ sztuk.

$A(i)$ — największa wartość jaką można uzyskać za pomocą plecaka o pojemności i.

Mamy wtedy zależność rekurencyjną:

$$\begin{aligned} A(0) &= 0 \\ A(i) &= \max\{c_j + A(i - w_j) : w_j \leq i\} \end{aligned}$$

Wyznaczenie kolejno: $A(0), A(1), \ldots, A(W)$ rozwiązuje zadanie w złożoności $O(WN)$

```
PlecakDynamiczny1(W, N, w, c) {
  for (i=0 to W )
    A[i] = 0;

  for (i=1 to W )
    for (j=1 to N )
      //czy j-ty towar mieści się w plecaku o rozmiarze i
      if ( w[j] <= i )
        A[i] = max(A[i], A[i-w[j]] + c[j]);
}
```

Wariant II:

Załóżmy, że każdy towar można **wziąć lub nie wziąć**.

$A(i,j)$ — największa wartość jaką można uzyskać za pomocą pierwszych i towarów i plecaka o pojemności j.

Mamy wtedy zależność rekurencyjną:

$$
\begin{array}{rcll}
A(0,j) & = & 0 & \\
A(i,0) & = & 0 & \\
A(i,j) & = & A(i-1,j) & \text{jeśli } w_i > j \\
A(i,j) & = & \max(\underbrace{A(i-1,j)}_{bez\ \ i}, \underbrace{c_i + A(i-1,j-w_i)}_{z\ \ i}) & \text{jeśli } w_i \leq j
\end{array}
$$

```
PlecakDynamiczny2(W, N, w, c) {
  for (i=0 to N )
    A[i,0] = 0;
  for (j=0 to W )
    A[0,j] = 0;

  for (i=1 to N )
    for (j=0 to W )
      if ( w[i] > j )
        A[i,j] = A[i-1,j];
      else
        A[i,j] = max(A[i-1,j], A[i-1,j-w[i]] + c[i]);
}
```

Złożoność: $O(NW)$

Dyskretny problem plecakowy II

- Mamy plecak o pojemności W.
- Mamy dany zbiór N towarów. Każdy towar ma swoją cenę c_i i wielkość w_i.
- Każdy towar w_i można wziąć lub nie wziąć.
- Suma wielkości załadowanych towarów musi być równa W.
- Wartość załadowanego plecaka ma być możliwie największa.

Co rozwiązuje problem?
Pełny przegląd możliwości: $O(2^N)$
Problem NP zupełny!

Języki formalne

- Alfabet — zbiór symboli (liter) Σ
  np.: $\Sigma = \{0, 1\}$
- ϵ — słowo puste
- Σ^* — zbiór wszystkich słów
  np.: $\Sigma^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, \ldots\}$ dla Σ j.w.
- Język to podzbiór Σ^* dla ustalonego zbioru symboli Σ
  np.: $L = \{1, 11, 111, 1111, \ldots\}$,
  np.: $L = \{10, 1100, 111000, 11110000, \ldots\}$
- Na językach można wykonywać: sumy, przecięcia, dopełnienie, domknięcie Kleene'a (L^*)
  $L^* = \{\epsilon\} \cup L \cup L^2 \cup L^3 \cup \ldots$, gdzie
  L^k to k-krotna konkatenacja L z L.

Język i Problem decyzyjny

Niech Q będzie problemem decyzyjnym:

$$Q : \Sigma^* \to \{0, 1\}$$

Wtedy możemy zdefiniować język:

$$L = \{x \in \Sigma^* : Q(x) = 1\}$$

Algorytm weryfikacji

definiowany jest jako algorytm o dwóch argumentach.
Pierwszy to zwykłe wejście x (ciąg danych), a drugie to *świadectwo*.
Algorytm weryfikuje ciąg x jeśli istnieje świadectwo y takie, że $A(x, y) = 1$, czyli algorytm A weryfikuje x z pomocą świadectwa y.
Język L weryfikowany przez algorytm A to:

$$L = \{x \in \{0,1\}^* \ : \ \text{istnieje } y \in \{0,1\}^*, \text{takie, że } A(x, y) = 1\}$$

L składa się z takich słów x, dla których istnieje y, które użyte przez A daje mu możliwość weryfikacji przynależności x do L.
Jeśli jakiś $x \notin L$, to nie istnieje y za pomocą którego A zweryfikowało by pozytywnie x (a podsunięcie fałszywego świadectwa y dla A też kończy się klęską! Ponieważ A potrafi weryfikować).

Problem cyklu Hamiltona

Cykl Hamiltona dla grafu nieskierowanego G to cykl prosty zawierający wszystkie wierzchołki z V.
Problem cyklu Hamiltona to sprawdzenie czy G ma cykl Hamiltona.
Można zdefiniować język formalny:

$$GCH = \{\langle G \rangle \ : \ G \text{ jest grafem z cyklem Hamiltona}\}$$

jako zbiór słów–grafów, które posiadają cykl Hamiltona.

- Zadanie algorytmu można wtedy widzieć jako sprawdzenie czy dany graf G należy do języka GCH, czyli czy posiada cykl Hamiltona.

- Natomiast zadaniem algorytmu weryfikacji jest sprawdzenie czy dla grafu G i pewnego świadectwa y, G jest grafem Hamiltonowskim. W przypadku problemu grafu Hamiltonowskiego (zawierającego cykl H.) *świadectwem* może być lista wierzchołków cyklu H. (jeśli istnieje ...). Algorytm weryfikacji wtedy ma łatwiejsze zadanie, weryfikuje tylko czy *świadectwo* jest prawdziwym cyklem dla danych, czyli grafu G. Co jest już prostym zadaniem (o złożoności $O(n^2)$).

Klasy złożoności, algorytmy weryfikacji i problemy nierozstrzygalne

Klasy złożoności problemów:

- *P* — problemy, dla których daje się skonstruować algorytm o złożoności wielomianowej ($O(n^k)$, k–const) względem rozmiaru zadania n. Problemy te nazywa się łatwymi.
- *NP* — *intuicyjnie:* problemy, dla których nie są znane algorytmy o złożoności wielomianowej ($O(n^k)$, k–const) względem rozmiaru zadania n.
  Formalnie: Problemy dla których istnieje algorytm weryfikacji działający w czasie wielomianowym.

Problemy NP–zupełne i NP–trudne

- *NP*–zupełne. Podzbiór problemów *NP*, do których dają się redukować inne problemy klasy NP w czasie wielomianowym. *Redukować L_1 do L_2 ($L_1 \leq_P L_2$)* oznacza przekształcać jeden problem w drugi w pewien formalny i algorytmizowalny efektywnie (czas wielomianowy) sposób.

  Formalnie L jest *NP*–zupełny jeśli:

  1. $L \in NP$
  2. $\forall_{L' \in NP} \;\; L' \leq_P L$
- NP–trudne — problemy które spełniają założenie 2 ale niekoniecznie założenie 1.

Klasy złożoności cd.

- Nie jest znana odpowiedź na pytanie: Czy $P = NP$?
- Wiemy, że $P \subseteq NP$
- Wiemy, że $NPC \subseteq NP$
- Większość sądzi, że $P \neq NP$.
- Z innej strony gdyby się nawet okazało, że $P = NP$, to stopień wielomianu funkcji złożoności byłby i tak bardzo wielki (tak że algorytm byłby niepraktyczny...).

Problemy nierozstrzygalne

Problem nierozstrzygalny to taki problem dla którego nie tylko nie ma szybkiego algorytmu, ale wręcz nie istnieje żaden algorytm, który rozwiązuje dany problem.

Czy ktoś zna taki problem?

- Problem stopu:

  Odpowiedzieć na pytanie:
  Czy dla algorytmu/programu A i danych x algorytm/program skończy swe działania.

- Problem domina:

  Mając skończony zbiór wzorów kafelków i nieograniczoną ilością kafli każdego wzoru odpowiedzieć na pytanie: Czy da się takimi kaflami pokryć dowolną powierzchnię zachowując kolory kafli na styku krawędzi.

  Wąż–domino (płaszczyzna – rozstrzygalny, półpłaszczyzna – nierozstrzygalny, obszar ograniczony – rozstrzygalny)

- Problem równoważności składniowej dwóch języków.

Inne problemy NP-zupełne

- Problem spełnialności reguł logicznych budowanych z bramek układów (AND, OR, XOR, NOT), czy także spełnialność reguł logicznych
- Problem ścieżki i cyklu Hamiltona
- Problem komiwojażera
- Problem kliki
- 3-kolorowania grafu
- Problem plecakowy na pełne wypełnienie określonej objętości podzbiorem towarów.

Problem komiwojażera (problem NP-zupełny) i algorytmy aproksymacyjne

- Znaleźć cykl o najmniejszym koszcie, który jest określony przez sumę wag krawędzi cyklu Hamiltona.

Można udowodnić, że problem jest *NP*–zupełny pokazując, że inne problemy są redukowalne do znajdowania cyklu Hamiltona. To można sprowadzić do redukcji znajdowania cyklu Hamiltona do p. komiwojażera.

- Dla problemów *NP*–zupełnych często by rozwiązać je efektywnie poszukuje się algorytmów których rozwiązanie nie jest optymalne lecz przybliżone do optymalnego w pewnym stopniu. Algorytmy aproksymacyjne to cała gałąź algorytmów, których celem jest poszukiwanie efektywnych algorytmów przybliżających rozwiązania problemów *NP*–trudnych możliwie jak najlepiej, i nie rzadko z pewnymi gwarancjami.

- Dla praktycznych problemów, gdy wierzchołki są punktami przestrzeni Euklidesowej koszt krawędzi może być zdefiniowany przez zwykłą odległość Euklidesową.
- Można skorzystać z algorytmu Primy do wyznaczenia minimalnego drzewa rozpinającego. Wtedy algorytm może wyglądać jak poniżej.

```
TSP_Prima(G,c)
    r= wybierz wierzchołek korzeń;
    T=MST_Prima(G,r);
    niech L będzie listą wierzchołków
        drzewa rozpinającego T w kolejności preorder
    return cykl złożony z wierzchołków znajdujących
        się w kolejności jak w L.
end
```

Preorder wyświetla wierzchołki zgodnie z kolejnością odwiedzin danego wierzchołka w drzewie T.

- Algorytm TSP_Prima znajduje cykl, który jest niedłuższy niż 2-krotność optymalnej drogi komiwojażera.

Wyszukiwanie wzorca

Dane: alfabet Σ, tablica $T[1..n]$, tablica–wzorzec $P[1..m]$, wartości tablic T i P są elementami alfabetu Σ.
Problem: Czy P jest podciągiem T, tj. czy istnieje s takie, że:

$$T[s+1..s+m] = P[1..m]$$

$$T[s+i] = P[i] \quad i = 1, \ldots, m$$

Prosty algorytm wyszukiwania wzorca

```
1 procedure SzukajWzoraca(P,T) {
2     n=lenght(T);
3     m=length(P);
4     for (s=0 to n−m)
5         if (T[s+1..s+m] = P[1..m] )
6             return s;
7     return −1;            {brak wzorca}
8 }
```

Czas działania wiersza 5 — $\Theta(m)$. Implementacja powinna sprawdzać do pierwszej różnej pary.
Łącznie czas: $\Theta((n-m+1)m)$.

Algorytm Rabina-Karpa I

- Bardzo dobrze zachowuje się średnio (w praktyce)
- Choć jego pesymistyczna złożoność też wynosi $O((n-m+1)m)$
- Załóżmy, że na alfabet Σ składają się cyfry $0, 1, \ldots, 9$ (algorytm łatwo uogólnić na dowolny alfabet).
- Wtedy $T[s+1..s+m]$ można interpretować jako liczbę dziesiętną t_s (dla innego alfabetu może to być liczba w pewnym innym systemie). Podobnie $P[1..m]$ może być widziane jako liczba dziesiętna p.
- Wartości takich liczb można wyznaczać w czasie $O(m)$:

$$p = P[m] + 10(P[m-1] + 10(P[m-2] + \ldots + 10(P[2] + 10P[1]) \ldots))$$

Algorytm Rabina-Karpa II

- Łatwo zauważyć, że na podstawie t_s można szybko obliczyć t_{s+1}:

$$t_{s+1} = 10(t_s - 10^{m-1}T[s+1]) + T[s+m+1]$$

Aby uogólnić wystarczy w miejsce 10 wpisać podstawę d.
Np.:
$t_s = 1234$,
$m = 4$,
$T[s+1] = 5$,
wtedy
$t_{s+1} = 10(1234 - 10^3) + 5 = 2345$

- Warto zauważyć, że koszt wyznaczenia p to $O(m)$, także koszt t_i to $O(m)$. Choć koszt wyznaczenia wszystkich t_i ogranicza się do $O(n+m)$.
- Problem gdy m jest nie (bardzo) małe, a to trzeba wkalkulować.
- Rozwiązaniem jest liczenie p i t_s modulo q dla odpowiednio dobranej wartości q.
- Po co? Jeśli $(p \% q) \neq (t_s \% q)$ to nie ma szans, aby $P = T[s+1..s+m]$!
- Jak wybrać q?
  Liczba pierwsza, dla której $10q$ mieści się w słowie komputera.
  Dlaczego tak?

- Co daje nowy wzór na t_{s+1}:

$$t_{s+1} = (d(t_s - h \cdot T[s+1]) + T[s+m+1]) \ \% \ q,$$

gdzie $h = d^{m-1} \ \% \ q$ – odpowiada wartość jedynki na najwyższej pozycji (porównaj wzór poprzedni).

- UWAGA(!) Jednak po takich zmianach to, że $t_s = p \ \% \ q$ nie implikuje, że $P = T[s+1..s+m]$. W takim przypadku trzeba jednak sprawdzić czy rzeczywiście $P = T[s+1..s+m]$ tradycyjnie...

```
SzukajWzoraca_Rabin_Karp(P,T,d,q)
{
    n=length(T);
    m=length(P);
    h=d^(m-1) % q;
    p=0;
    t_0=0;


    for (i=1 to m)
    {
        p=(dp+P[i]) % q;
        t_0=(dt_0+T[i]) % q;
    }


```

```
    for (s=0 to n−m )
    {
        if (p==t_s )
            if (T[s+1..s+m] == P[1..m] )
                return s;
        if (s<n−m )
            t_{s+1}=(d(t_s−h∗T[s+1]) +T[s+m+1]) % q;
    }
    return −1;          {brak wzorca}
}
```

Złożoność

w.4 i 8–12 — $O(m)$
w.18 — $O(m)$

Razem: $O((n - m + 1n)m)$.

Natomiast oczekiwany czas tego algorytmu to $O(n + m)$.
Algorytm o wiele bardziej godny polecenia!!!

Algorytm Knutha–Morrisa–Pratta

- W stronę algorytmu o złożoności $\Theta(n + m)$.
- Funkcja prefiksowa wzorca P.

Cel: unikanie *pustego/straconego* szukania stosowanego w algorytmie naiwnym.

```
T = b a c b a b a b a a b c b a b
P =         a b a b a c a
    -- s ---           x
            --- q ---

T = b a c b a b a b a a b c b a b
P =             a b a b a c a
   ----- s' ----        x
                - k -

            a b a b a
                a b a
```

Szukanie minimalnego przesunięcia s' większego od s, dla którego:

$$P[1 \,..\, k] = T[s' + 1 \,..\, s' + k]$$

$s' + k = s + q$, q jest równe długości wspólnego prefiksu dla P i $T[s + 1 \,..\, s + m]$.

- Właśnie od takiego przesunięcia warto kontynuować dalsze szukanie wzorca.
- Co więcej pierwszych k znaków $T_{s'}$ stanowi suffiks P_q ($P_q = q$ pierwszych znaków P).
- Czyli $s' = s + (q - k)$ to dobry skok!
- Funkcję prefiksingu definiujemy poprzez:

$$\pi[q] = \arg\max_{k<q} \; P_k \sqsupset P_q$$

  $P_k \sqsupset P_q$ oznacza, że P_k jest suffiksem P_q (dla $\sqsubset$ – prefixem).
- Jak widać $\pi[q]$ zależą tylko od P!
- i $\pi[q] < q$

i	1	2	3	4	5	6	7	8	9	10
P[i]	a	b	a	b	a	b	a	b	c	a
pi[i]	0	0	1	2	3	4	5	6	0	1

Dla $q = 8, 6, 4, 2$ otrzymujemy:

P_8	a	b	a	b	a	b	a	b	c	a										
P_6			a	b	a	b	a	b	a	b	c	a								
P_4					a	b	a	b	a	b	a	b	c	a						
P_2							a	b	a	b	a	b	a	b	c	a				
P_0									a	b	a	b	a	b	a	b	a	b	c	a

$$\pi^*[q] = \{q, \pi[q], \pi^2[q], \ldots, \pi^t[q]\} \qquad \pi^i[q] = \begin{cases} q & i = 0 \\ \pi[\pi^{i-1}[q]] & i > 0 \end{cases}$$

$$\pi^*[8] = \{8, 6, 4, 2, 0\}$$

$\pi^*[q]$ — zbiór prefiksów P będących sufiksami P_q

$$\pi^*[q] = \{k : P_k \sqsupset P_q\}$$

$$E_{q-1} = \{k : \ k \in \pi^*[q-1] \ \wedge \ P[k+1] = P[q]\}$$

Oczywiście: $E_{q-1} \subseteq \pi^*[q-1]$

E_{q-1} zbiór takich k, że $P_k \sqsupset P_{q-1}$ i $P_{k+1} \sqsupset P_q$ — czyli P_k daje się rozszerzyć do P_{k+1} i jest sufiksem P_q.

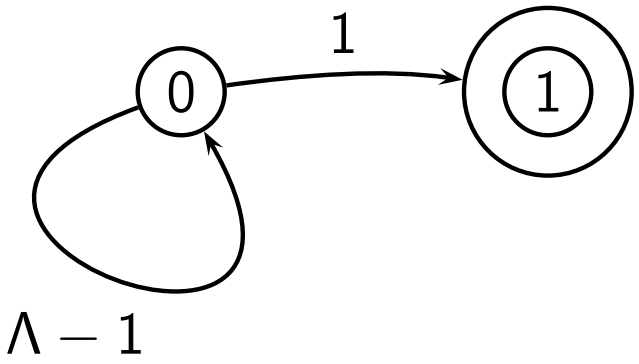
$$\pi[q] = \begin{cases} 0 & E_{q-1} = \emptyset \\ 1 + \max\{k \in E_{q-1}\} & \neg \end{cases}$$

```
SzukajWzoraca_KMP(P,T) {
    n=lenght(T);
    m=length(P);
    π=Compute_Prefix(P);
    k=0;
    for (q=1 to n )
    {
        while (k>0 && P[k+1]!=T[q] )
            k=π[k];
        if (P[k+1]==T[q] )
            k=k+1;
        if (k==m )
            Output q−m; // return q-m;
            k=π[k];
    }
}
```

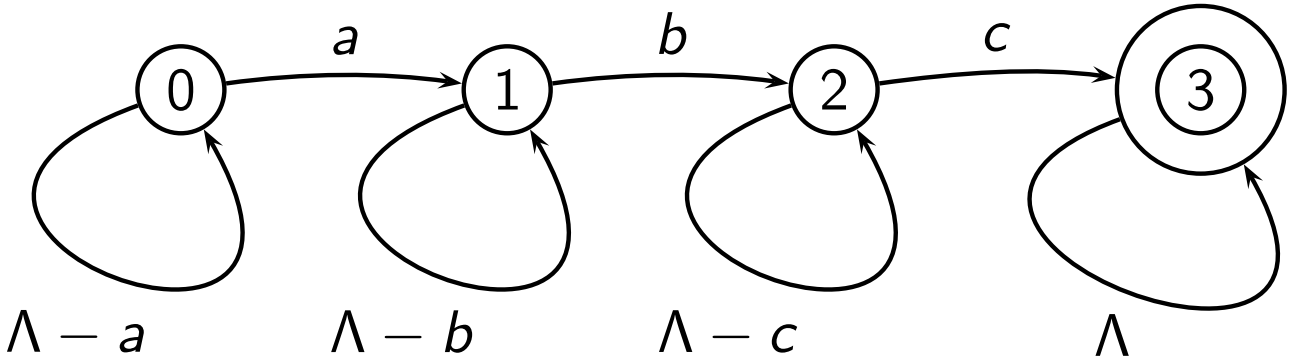
```
Compute_Prefix(P) {
    m=length(P);
    π[1]=0;
    k=0;
    for (q=2 to m )
    {
        while (k>0 && P[k+1]!=P[q] )
            k=π[k];
        if (P[k+1]==P[q] )
            k=k+1;
        π[q]=k;
    }
    return π;
}
```

Automaty skończone

- Automaty skończone reprezentowane są poprzez grafy o skończonej liczbie wierzchołków i pewnej liczbie krawędzi.
- Wierzchołki odpowiadają stanom automatu a krawędzie odpowiadają za przejścia pomiędzy stanem z którego prowadzi dana krawędź i stanem do którego prowadzi ta krawędź. Krawędzie są etykietowane pojedynczymi literami bądź podzbiorem liter pewnego alfabetu Λ.



- Działanie automatu polega na przechodzeniu przez stany automatu zgodnie ze słowem wejściowym (dane wejściowe) i układem stanów i przejść pomiędzy nimi (czyli krawędziami w grafie).
- Automat zaczyna pracę w stanie początkowym (często oznaczonym przez 0).
- Następnie zgodnie z pierwszą literą wyrazu następuje przejście ze stanu zerowego do takiego stanu, do którego prowadzi krawędź zaetykietowana przez pierwszą literę wyrazu. Po czym kontynuowany jest taki proces, czyli pobierana jest następna litera wyrazu i automat przechodzi do takiego stanu, do którego da się przejść właśnie z taką literą alfabetu.

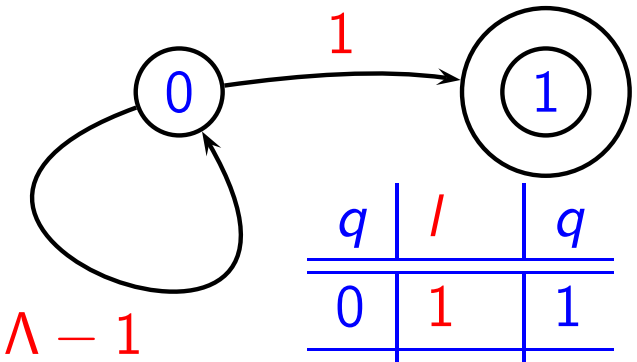


- Automat kończy pracę, gdy przeanalizuje całe zadane słowo wejściowe lub zabraknie przejść dla danych par stan–litera (automat przechodzi z pewnego stanu q z literą l). Gdyby zabrakło przejść słowo wejściowe nie będzie zaakceptowane.
- Stany automatu dzielimy na stany akceptujące i nieakceptujące.
- Zadaniem automatu jest zaakceptowanie bądź nie zaakceptowanie danego słowa wejściowego. Mianowicie, gdy automat skończy pracę w stanie akceptującym oznacza to, że słowo jest akceptowane (rozpoznawane) przez automat skończony. W przeciwnym wypadku słowo nie jest akceptowane (nie jest rozpoznawane).
- Każdy automat skończony rozpoznaje wyrazy, które należą do pewnego języka L nad alfabetem Λ.

$$A = \{Q, \Lambda, \delta, q_0, F\}$$

Q	zbiór stanów
Λ	alfabet (zbiór symboli)
$\delta : Q \times \Lambda \to Q$	funkcja przejścia
q_0	stan początkowy
$F \subseteq Q$	zbiór stanów akceptujących

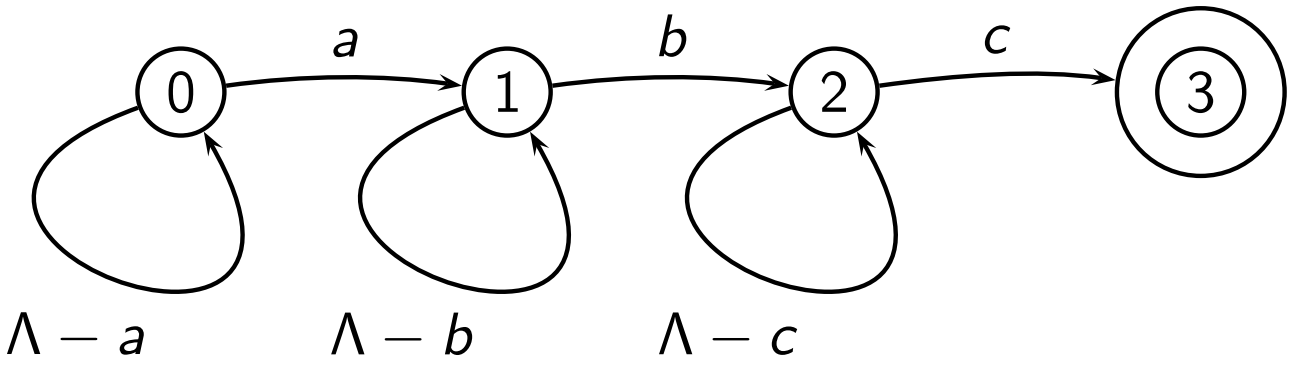
Q to wierzchołki grafu
δ definiuje krawędzie:



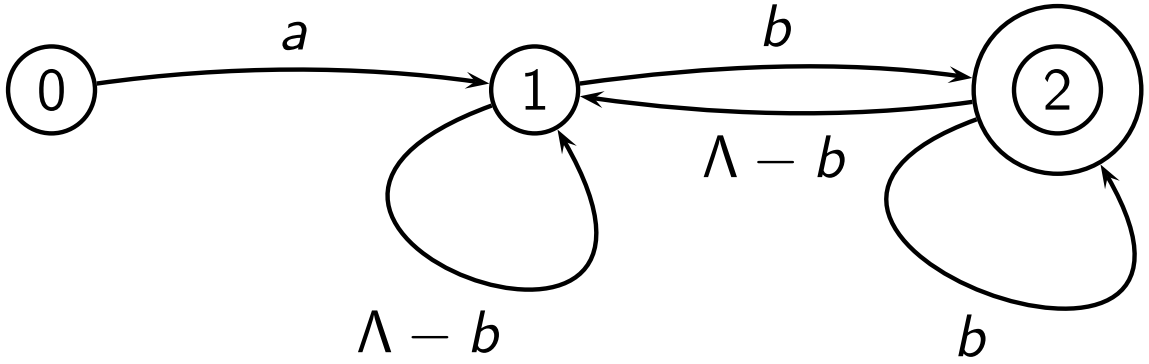
q	l	q
0	1	1
0	$\neg 1$	0

słowa w których występuje a później jest też b i później jest i c

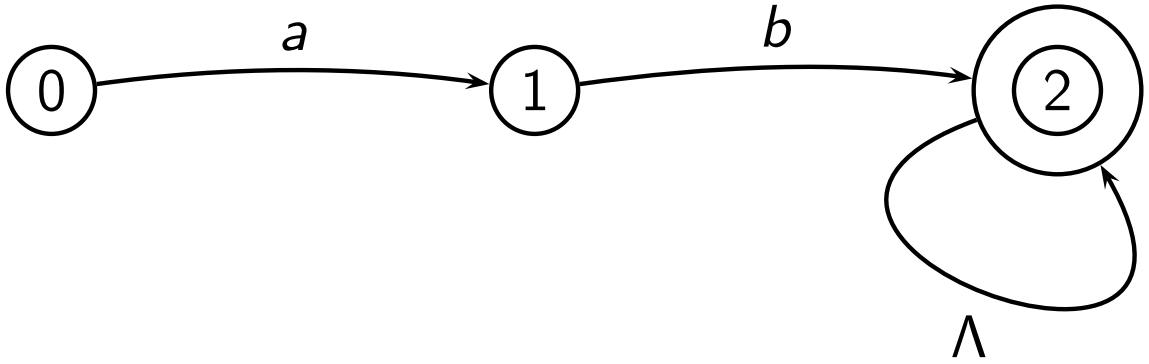
....a....b....c



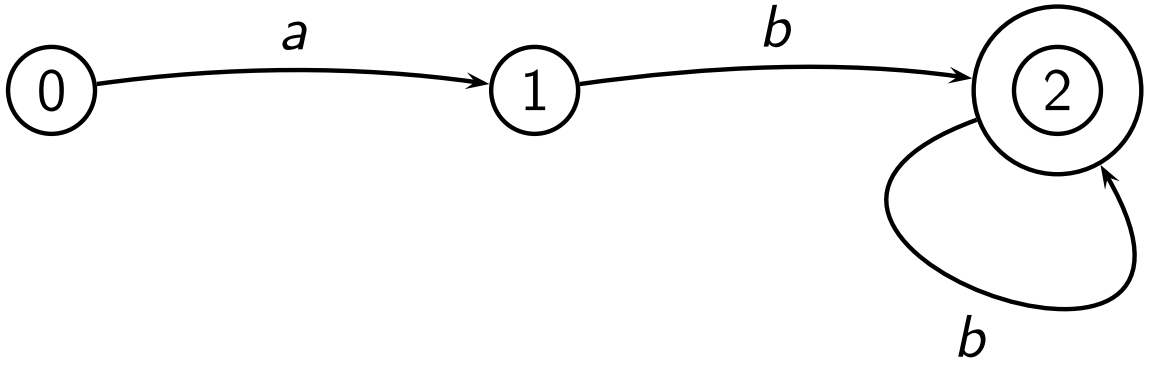
słowa, które rozpoczynają się na a i kończą na b



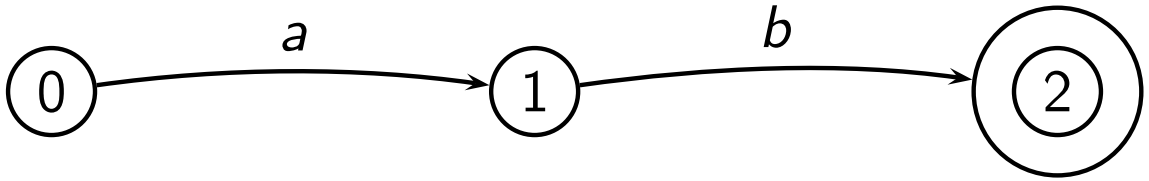
wyrazy rozpoczynające się od *ab*



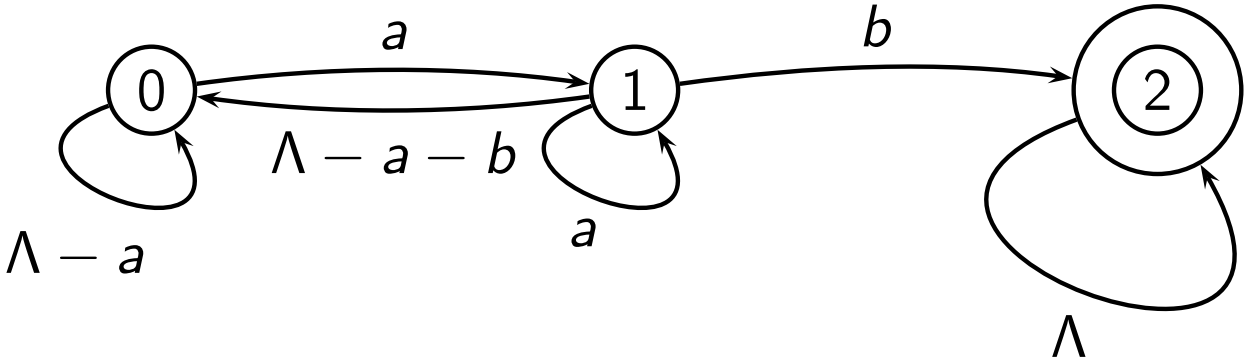
wyrazy rozpoczynające się od *a* a następnie mają jedno lub więcej *b*

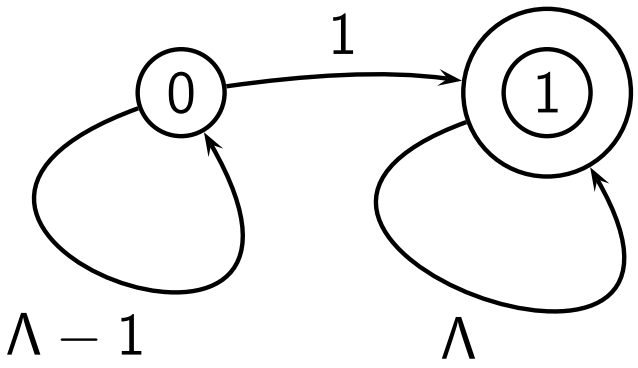


słowo ab (tylko ab!)

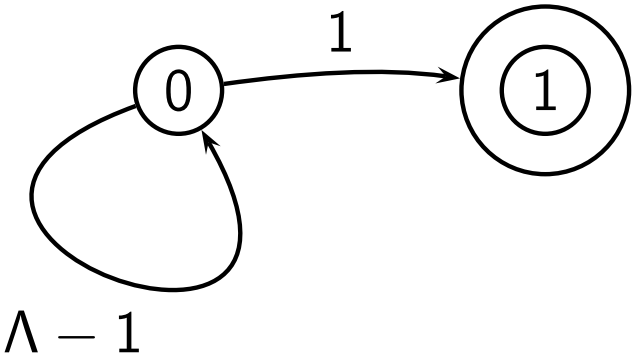


słowo zawierające choć jedno wystąpienie ciągu ab

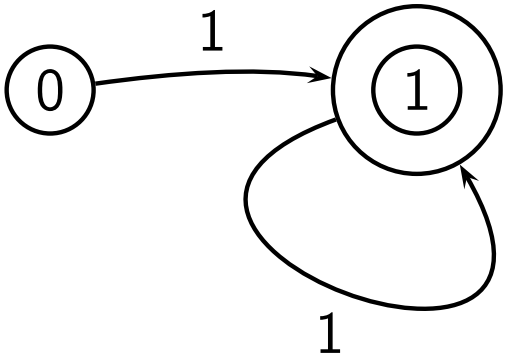




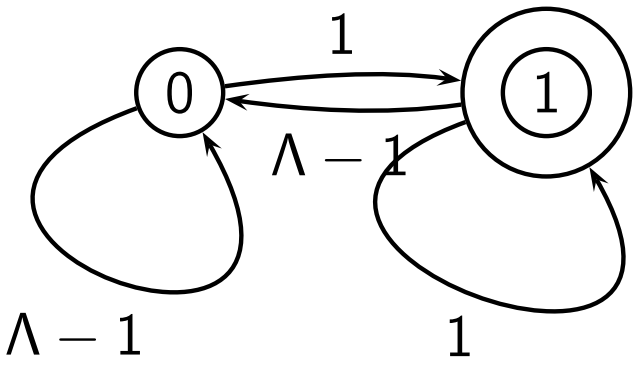
słowa z jedynką



s. z jedynką tylko na końcu



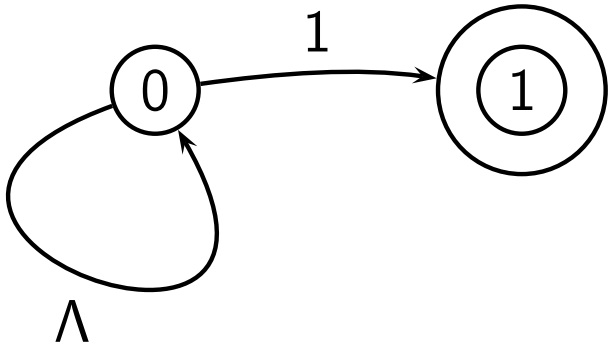
s. z jedynek



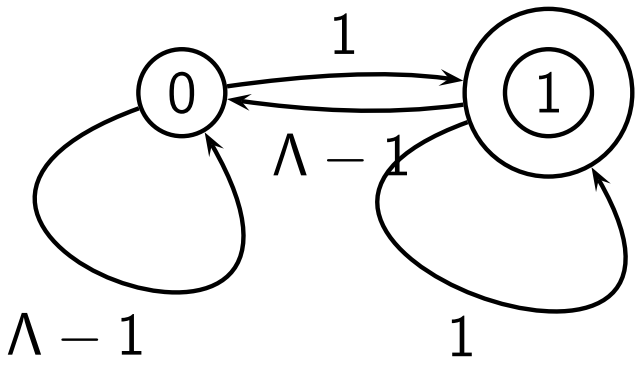
s. z jedynką na końcu

Automaty niedeterministyczne

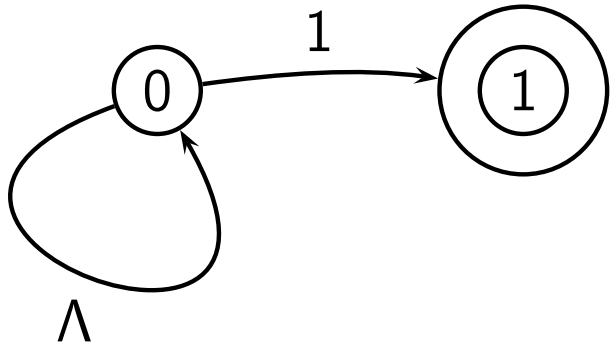
W automatach niedeterministycznych (w odróżnieniu od deterministycznych) może istnieć więcej niż jedno przejście dla pary stan–litera.



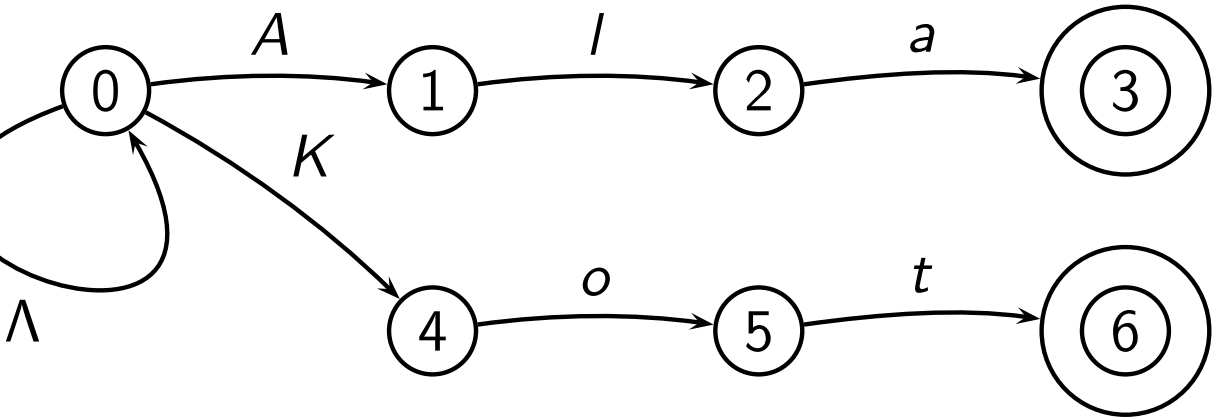
- Oznacza to, że automat niedeterministyczny może przechodzić faktycznie do jednego lub większej ilości stanów. Z których znów może dać się przejść na (łącznie) pewną liczbę sposobów, co jest wykonywane.
- Zakończenie pracy automatu niedeterministycznego jest zdefiniowane tak samo jak automatu deterministycznego.
- Automat niedeterministyczny akceptuje wyraz jeśli choć jeden ze stanów, w których automat zakończył pracę jest stanem akceptujący.



[DA] — s. z jedynką na końcu



[NDA] — s. z jedynką na końcu



- UWAGA: Każdy automat niedeterministyczny można zamienić na deterministyczny (tj. akceptujący słowa tego samego języka L).

Wyrażenia regularne

Wartością wyrażenia regularnego r jest pewien język $L(r)$.
Mówimy także, że język $L(r)$ składa się ze słów *pasujących* do wyrażenia r).

- Oznaczenia:

ϵ — pusty ciąg znaków
$\emptyset$ — zbiór pusty ciągów znaków
. — dowolny znak

- Operatory

suma — $a|b$

[a|k|p — a, k, p

ala|kot|pies — ala, kot, pies]

złożenie/konkatenacja — ab

[ala, kot,

(Ala|Ela)ma(kota|psa) — Alamakota, Elamakota, Alamapsa, Elamapsa]

domknięcie — a*

[a* — ϵ, a, aa, aa, aaa, ...

(a|b)* — ϵ, a, b, aa, ab, aaab, babab, ...]

- Kolejność realizacji operatorów

- domknięcie
- złożenie
- suma

Przykłady

0|1|2|3|4|5|6|7|8|9	cyfra
(a|b|...|z)*	wyraz zbudowany z liter
ab.*	ciąg znaków rozpoczynających się od *ab*
(a|b).*	ciąg znaków rozpoczynających się od *a* lub *b*
.*ab.*	ciąg znaków zawierający *ab*
.*(ab|AB).*	ciąg znaków zawierający *ab* lub *AB*

(_|a|A|b|B|...|z|Z)(_|0|1|...|9|a|A|b|B|...|z|Z)* — definicja identyfikatora akceptowanego przez języki C i C++

Od wyrażenia regularnego do automatu

Czyli jak samemu zaimplementować obsługę wyrażeń regularnych.

- Z wyrażenia regularnego tworzymy automat skończony niedeterministyczny.
  Trzeba stworzy reguły zamian elementów wyrażenia regularnego na elementy automatu dla każdego z operatorów.
- Z automatu niedeterministycznego tworzymy automat deterministyczny.

- Bardzo przydatne!!! Patrz Unix, egrep, emacs, awk, pearl

Kopce

- H = Make-heap()
- Insert (H, x) — wstawia węzeł x (z kluczem key[x]) do kopca H
- Minimum (H)
- Extract-Min(H) — zwraca wartość wskaźnika o minimalnym kluczu
- Union(H1, H2) — tworzy i zwraca nowy kopiec będący sumą kopców H1 i H2
- Decrease-key(H,x,k) — zmiana wartości węzła x na k w kopcu H
- Delete(H,x) — usunięcie węzła x z kopca H

Kopce — złożoności funkcji

Funkcja	Kopiec binarny*	Kopiec dwumianowy*	Kopiec Fibonacciego**
Make-heap	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Insert	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$
Minimum	$\Theta(1)$	$\Theta(\log n)$	$\Theta(1)$
Extract-min	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$
Union	$\Theta(n)$	$\Theta(\log n)$	$\Theta(1)$
Decrease-key	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$
Delete	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$

Operacje Decrease-key i Delete wymagają jako argumentu wskaźnika do węzła, nie tylko wartości. W przeciwnym razie trzeba szukać, co jest droższe.

* – koszt pesymistyczny

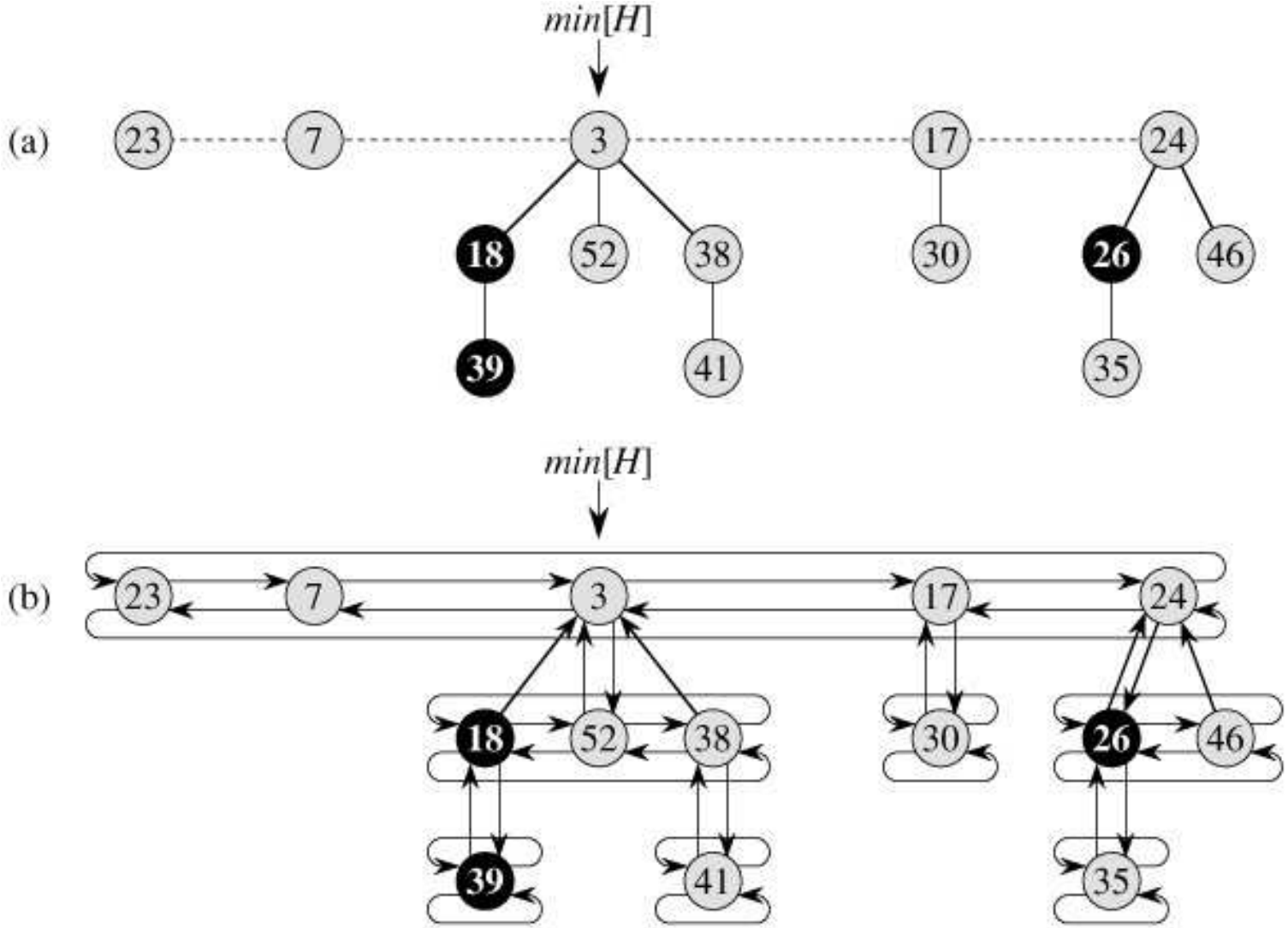
** – koszt zamortyzowany

Kiedy używać kopców innych niż binarne?

- Gdy liczba operacji Extract-min jest mniejsza Decrease-key.
- Gdy potrzebne są operacje Union na kopcach.

Po za tymi przypadkami nie warto stosować kopców bardziej wyrafinowanych niż binarne. Jednak w takich przypadkach, jak powyżej, radykalnie przyśpieszają potrzebne operacje.
Np.:
– Prima, najkrótsze drzewo rozpinające
– Dijkstra, najkrótsze ścieżki

Struktura kopców Fibonacciego



Tworzenie kopca i minimum

```
Fib_Make_heap(H)
{
    H.n = 0;
    H.min = NULL;
}
```

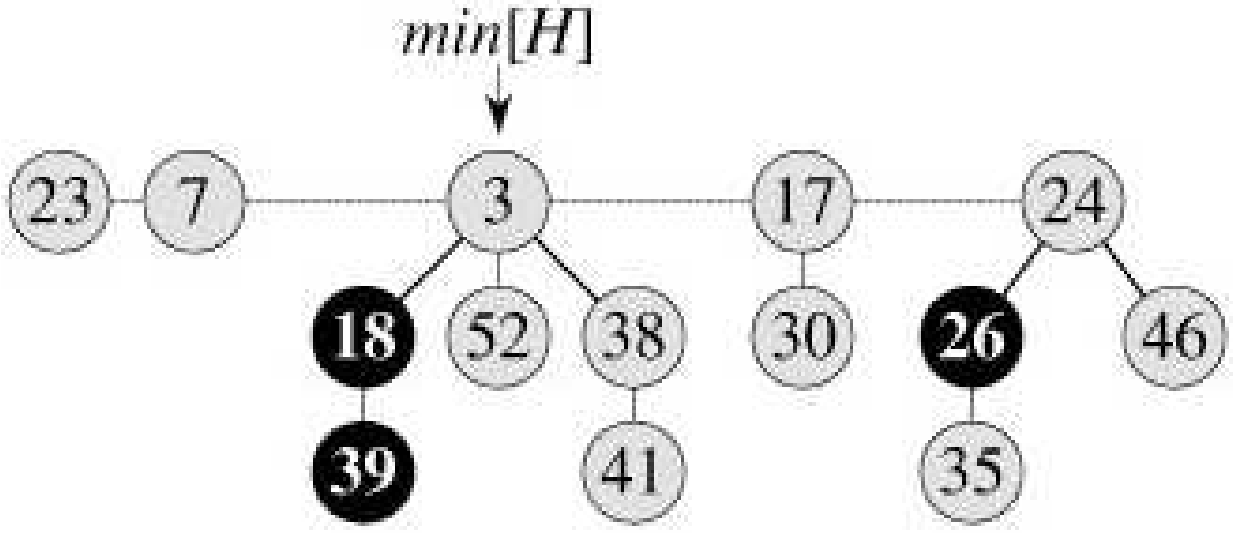
```
Minimum(H)
{
    return H.min;
}
```

Insert

```
FIB_HEAP_INSERT(H, x){
    x.degree = 0;
    x.p = NULL;
    x.child = NULL;
    x.left = x;
    x.right = x;
    x.mark = FALSE;
    concatenate the root list containing x with root list H;
    if (H.min == NULL || x.key < H.min.key)
        H.min = x;
    H.n++;
}
```

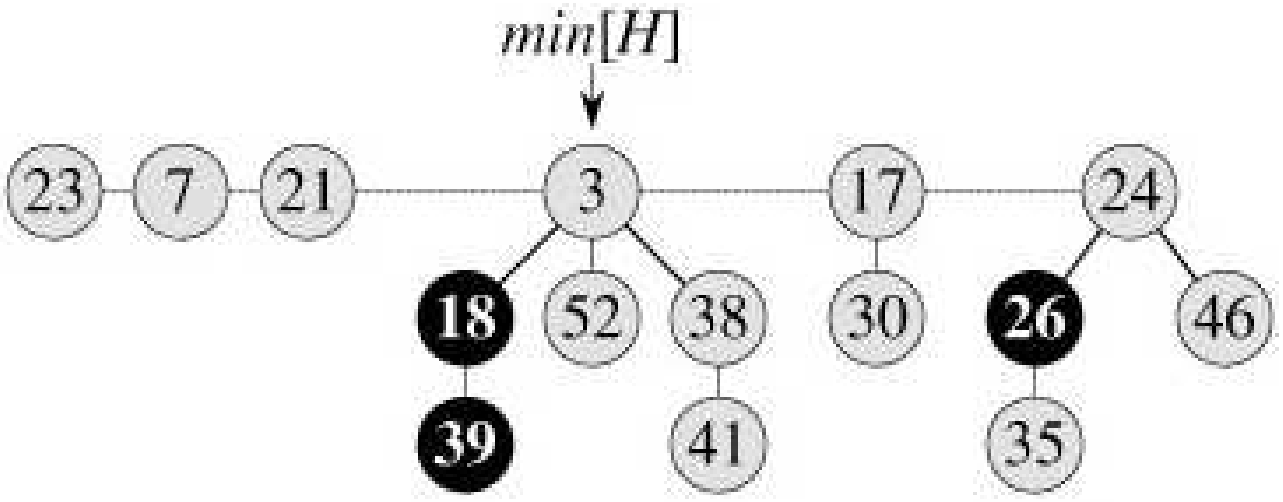
Dzięki konkatenacji list dwukierunkowych: $O(1)$.

$mark[x] == true \iff$ x stracił syna odkąd sam stał się dzieckiem



(a)

Po dodaniu 21:

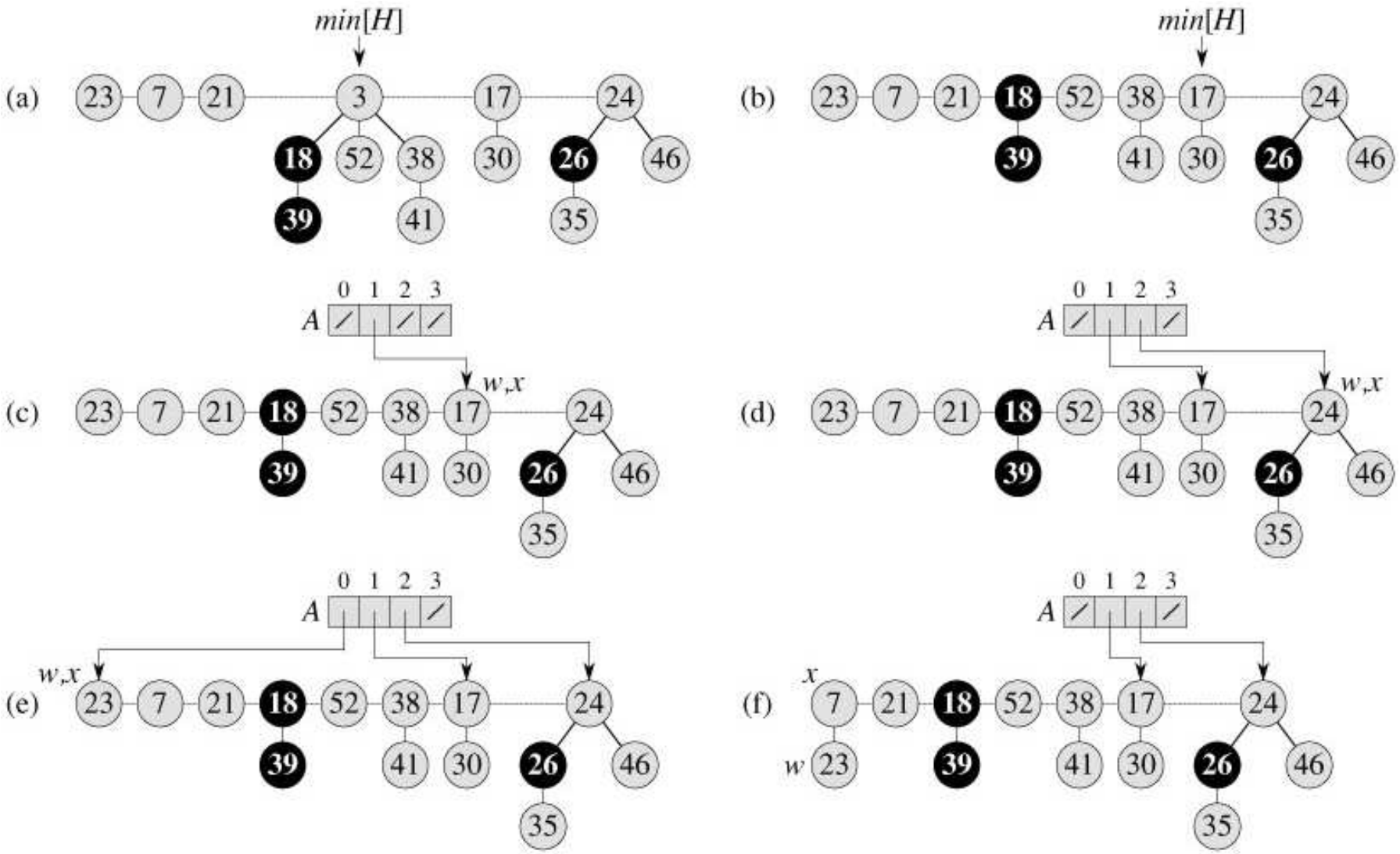


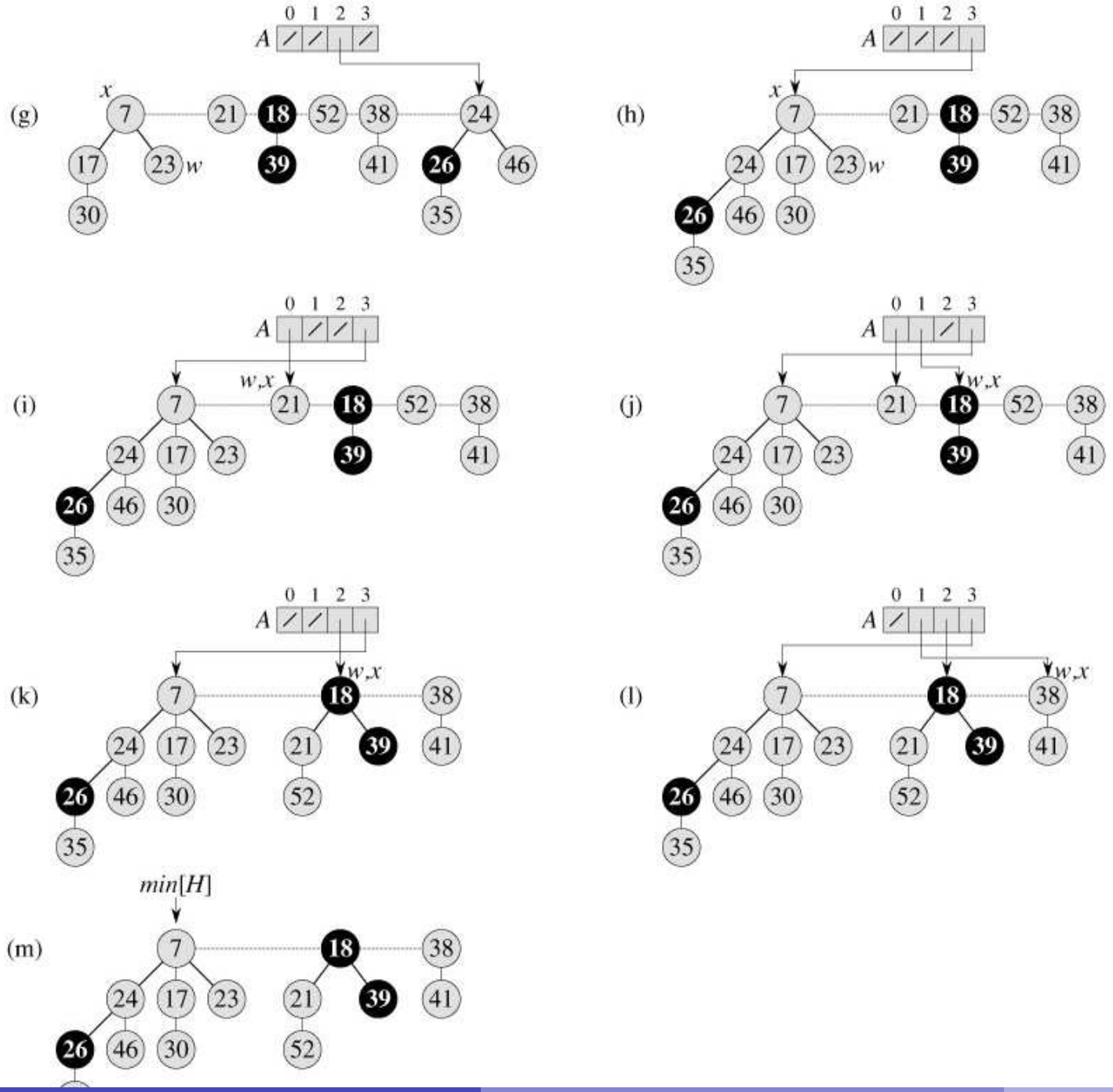
(b)

Łączenie kopców

```
FIB_HEAP_UNION(H1, H2) {
    H = MAKE_FIB_HEAP();
    H.min = H1.min;
    concatenate the root list of H2 with the root list of H;
    if ((H1.min == NULL) || (H2.min != NULL && H2.min < H1.min))
        H.min = H2.min;
    H.n = H1.n + H2.n;
    free the objects H1 and H2;
    return H;
}
```

```
FIB_HEAP_EXTRACT_MIN(H){
    z = H.min;
    if (z != NULL ){
        foreach (child x in z){
            add x to the root list of H
            p[x] = NULL;
        }
        remove z from the root list of H;
        if (z == z.right )      // z jedynym elementem H
            H.min = NULL;
        else{
            H.min = z.right;
            CONSOLIDATE(H);
        }
        H.n--;
    }
    return z;
}
```


(a)
min[H]
23 7 21 3 17 24
18 52 38 30 26 46
39 41 35
(b)
min[H]
23 7 21 18 52 38 17 24
39 41 30 26 46
35
(c)
A 0 1 2 3
w,x
23 7 21 18 52 38 17 24
39 41 30 26 46
35
(d)
A 0 1 2 3
w,x
23 7 21 18 52 38 17 24
39 41 30 26 46
35
(e)
A 0 1 2 3
w,x
23 7 21 18 52 38 17 24
39 41 30 26 46
35
(f)
A 0 1 2 3
x
w
7 21 18 52 38 17 24
23 39 41 30 26 46
35


(g)
(h)
(i)
(j)
(k)
(l)
(m)
A
0 1 2 3
x
w
w,x
min[H]
7
21
18
52
38
24
17
23
39
41
26
46
30
35

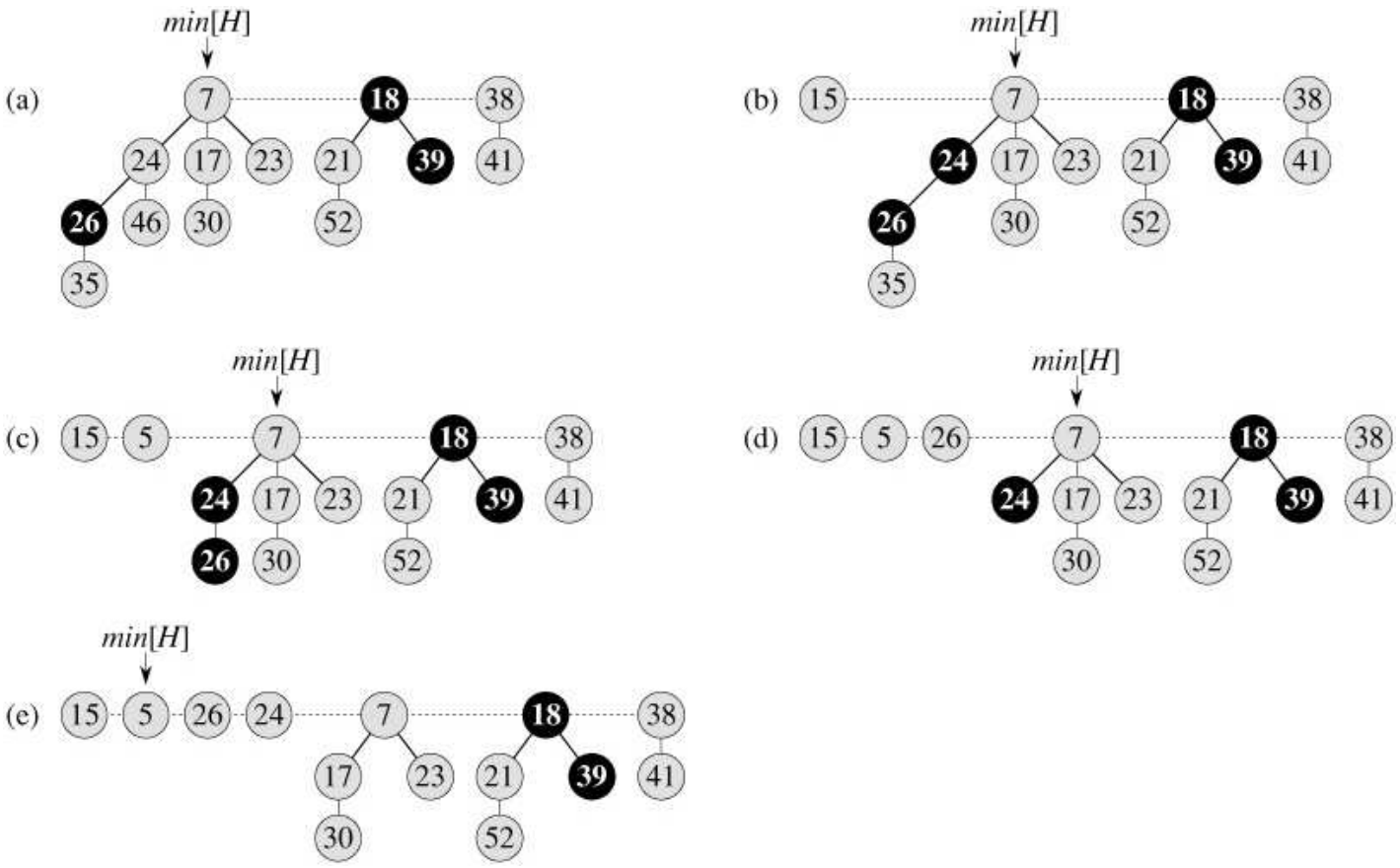
```
CONSOLIDATE(H){
    for (i = 0 to D(n[H])) A[i] = NULL;
    foreach( node w in the root list of H ){
        x = w;
        d = x.degree;
        while (A[d] != NULL ){
            y = A[d];        // Another node with the same degree as x.
            if (x.key > y.key) exchange x and y;
            FIB_HEAP_LINK(H, y, x); // y become a child of x
            A[d] = NULL;
            d++;
        }
        A[d] = x;
    }
    H.min = NULL;   // opróżnienie listy korzeni
    for (i = 0 to D(n[H]) )
        if (A[i] != NULL) {
            add A[i] to the root list of H
            if (H.min == NULL || A[i].key < H.min.key)
                H.min = A[i];
        }
}
```

```
FIB_HEAP_LINK(H, y, x){
    remove y from the root list of H
    make y a child of x
    x.degree++;
    y.mark = FALSE;
end
```

```
FIB_HEAP_DECREASE_KEY(H, x, k){
    if (k > x.key) error "new key is greater than current key"
    x.key = k;
    y = x.p;
    if (y != NULL && x.key < y.key){
        CUT(H, x, y);
        CASCADING_CUT(H, y);
    }
    if (x.key < H.min.key)
        H.min = x;
}
CUT(H, x, y){
    remove x from the child list of y,
    y.degree--;
    add x to the root list of H;
    x.p = NULL;
    x.mark = FALSE;
}
```

```
CASCADING_CUT(H, y){
    z = y.p;
    if (z != NULL )
      if (y.mark == FALSE)
        y.mark = TRUE;
      else{
        CUT(H, y, z);
        CASCADING_CUT(H, z);
      }
}
```

a–b) 46 → 15 b–e) 35 → 5



```
procedure FIB_HEAP_DELETE(H, x)
    FIB_HEAP_DECREASE_KEY(H, x, −∞)
    FIB_HEAP_EXTRACT_MIN(H)
end
```

Geometria obliczeniowa

```
1 typedef long double real;
2 const real EPS = 1E-9;
3 inline bool IsZero(real x)
4 {
5   return x >= -EPS && x <= EPS;
6 }
```

```
1  struct POINT
2  {
3     real x, y;
4    bool undef;
5    bool paral;
6    POINT(real wx=0, real wy=0) :x(wx), y(wy), undef(false),
7       paral(false) { }
8    bool operator==(const POINT &b) const {
9       if(IsZero(x-b.x) && IsZero(y-b.y)) return true;
10      return false;
11   }
12   bool operator!=(const POINT &b) const {
13      return !((*this)==b);
14   }
15   bool operator <(const POINT &p) const {
16      return x < p.x || (IsZero(x-p.x) && y < p.y);
17   }
18 };
```

```
// skalarny a-b z a-c
inline real skal(POINT &a, POINT &b, POINT &c)
{
  return (b.x−a.x) ∗ (c.x−a.x) + (b.y−a.y) ∗ (c.y−a.y);
}
inline real dist(POINT &a, POINT &b)
{
  return sqrtl(skal(a, b, b));
}
// a-c na a-b
inline real dlrzutu(POINT &a, POINT &b, POINT &c)
{
  return skal(a, b, c)/dist(a, b);
} // punktu c na prosta a-b
inline POINT rzut(POINT &a, POINT &b, POINT &c)
{
  real f = skal(a, b, c) / skal(a, b, b);
  return POINT(a.x + f ∗ (b.x−a.x), a.y + f ∗ (b.y−a.y));
}
```

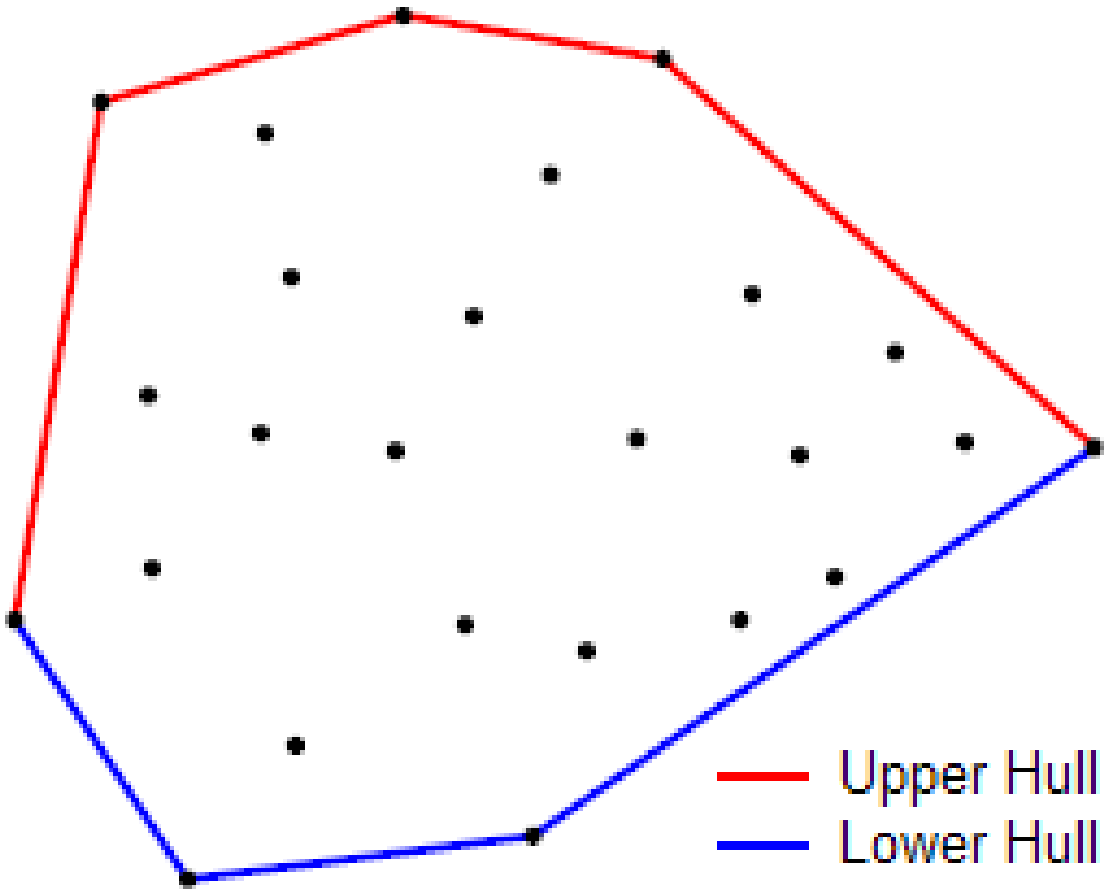
```
inline real det(POINT &a, POINT &b, POINT &c)
{
  return (b.x−a.x) ∗ (c.y−a.y) − (b.y−a.y) ∗ (c.x−a.x);
}
inline real det(POINT &a, POINT &b, POINT &c, POINT &d)
{
  return (det(a,d,b) + det(a,b,c))/2;
}
inline int sgn(real x)
{ return x < −EPS ? −1 : (x > EPS ? 1 : 0); }

inline int strona(POINT &a, POINT &b, POINT &c)
{
  return sgn(det(a, b, c));
}
inline real pole(POINT &a, POINT &b, POINT &c)
{
  return fabsl(det(a, b, c))/2;
}
```

```
inline POINT przeciecie_punktow(POINT &a, POINT &b,
    POINT &p, POINT &q)
{
  real p1 = det(a,b,p,q), p2 = det(a,b,p)/2, st = p2/p1;
  POINT res_p(0,0);
  if(IsZero(p1)) { res_p.undef=true; return res_p; }
  if(IsZero(p2*2 - p1)) { res_p.paral=true; return res_p; }
  return POINT(p.x + (q.x-p.x)*st, p.y + (q.y-p.y)*st);
}
// punkt przecięcia odcinków
inline POINT segments_intersection(POINT &a, POINT &b, POINT &c,
    POINT &d)
{
  POINT p = przeciecie_punktow(a,b,c,d);
  if(p.paral || p.undef)
    p.undef=true;
  if(segment(a,b,p) != 0 || segment(c,d,p) != 0)
    p.undef=true;
  return p;
}
```

```
1 // gdzie jest c na lini a-b: przed, za, pomiędzy
2 inline int segment(POINT &a, POINT &b, POINT &c)
3 {
4   if(skal(a,b,c)<0)
5     return -1;
6   else if(skal(b,a,c)<0)
7     return 1;
8   return 0;
9 }
```

Otoczka wypukła



```
convex-hull(point[] P)
    Sort P // względem x
    // będą przechowywały wierzchołki górnej i dolnej części
    U,L = empty lists;
    for( i=1 to n){
      while (L contains at least two points and the sequence of
         last two points of L and the point P[i] does not make
            a counter-clockwise turn)
         remove the last point from L;
      append P[i] to L; }
    for( i=n to 1 ){
      while (U contains at least two points and the sequence of
         last two points of U and the point P[i] does not make
            a counter-clockwise turn)
         remove the last point from U;
      append P[i] to U; }
    Remove the last point of each list
         (it is the same as the first point of the other list).
    Concatenate L and U to obtain the convex hull of P.
}
```

```
enum direction { left , right };

vector<POINT> convex_hull(vector<POINT> P) {
    int n = P.size(), k = 0;
    vector<POINT> H(2*n);
    // Sort points lexicographically
    sort(P.begin(), P.end());
    // Build lower hull
    for (int i = 0; i < n; i++) {
      while (k >= 2 && strona(H[k-2], H[k-1], P[i]) != right) k--;
      H[k++] = P[i];
    }
    // Build upper hull
    for (int i = n-2, t = k+1; i >= 0; i--) {
      while (k >= t && strona(H[k-2], H[k-1], P[i]) != right) k--;
      H[k++] = P[i];
    }
    H.resize(k);
    return H;
}
```

Drzewa sufiksowe

Link: suffix trees