

ALGORYTMY I STRUKTURY DANYCH

Spis treści

| | | |
|-----|---|----|
| 1. | Program komputerowy | 3 |
| 2. | Rekurencja a iteracja | 4 |
| 3. | Algorytm | 4 |
| 4. | Ciąg Fibonacciego | 4 |
| 5. | Tribanacji | 5 |
| 6. | Przykłady algorytmów | 5 |
| a. | Algorytm Euklidesa | 5 |
| b. | Min-Max | 6 |
| c. | Algorytmy sortujące | 7 |
| • | Stabilne | 7 |
| • | Niestabilne | 11 |
| 7. | Asymptotyczne tempo wzrostu | 13 |
| 8. | Metody obliczania rekurensji | 14 |
| 9. | Drzewa binarne | 15 |
| a. | Przechodzenie drzew BFS/DFS | 15 |
| b. | Drzewo poszukiwań binarnych – BST | 15 |
| c. | Kopiec | 15 |
| 10. | Algorytmy grafowe | 15 |
| a. | Graf | 15 |
| b. | Wzór Eulera | 15 |
| c. | Graf: pełny, zerowy, skierowany i nieskierowany | 15 |
| d. | Macierz sąsiedztwa | 15 |
| e. | Kolejka | 15 |
| f. | Przeszukiwanie wszerek BFS | 16 |
| g. | Przeszukiwanie w głąb DFS | 20 |
| h. | Algorytm Bellmana-Forda | 24 |
| i. | Algorytm DIJKSTRY | 29 |
| 11. | Zadania | 33 |

1. Program komputerowy

Jest to sekwencja symboli opisująca realizowanie obliczeń zgodnie z pewnymi regułami zwanymi językiem programowania. Program jest zazwyczaj wykonywany przez komputer, zwykle bezpośrednio, jeśli wyrażony jest w języku zrozumiałym dla danej maszyny lub pośrednio – gdy jest interpretowany przez inny program. Formalne wyrażenie metody obliczeniowej w postaci języka zrozumiałego dla człowieka nazywane jest kodem źródłowym, podczas gdy program wyrażony w postaci zrozumiałej dla maszyny nazywany jest kodem maszynowym bądź postacią binarną. Programy komputerowe można zaklasyfikować według ich zastosowań. Wyróżnia się zatem aplikacje użytkowe, systemy operacyjne, programy narzędziowe, gry wideo, kompilatory i inne. Natomiast programy wbudowane w urządzenia, przechowywane zwykle w pamięci flash, określa się jako firmware.

Kod źródłowy – zapis programu komputerowego przy pomocy określonego języka programowania, opisujący operacje jakie powinien wykonać komputer na zgromadzonych lub otrzymanych danych. Kod źródłowy jest wynikiem pracy programisty i pozwala wyrazić w czytelnej dla człowieka formie strukturę oraz działanie programu komputerowego. Przed wykonaniem kod źródłowy musi zostać poddany translacji na kod wynikowy, w procesie zwanym kompilacją. Polega on na konwersji kodu do postaci kodu wynikowego, najczęściej kodu maszynowego, jako jedynego możliwego do wykonania przez procesor.

Kod maszynowy (język maszyny) - to postać programu komputerowego (zwana postacią wykonywalną lub binarną) przeznaczona do bezpośredniego lub prawie bezpośredniego wykonania przez procesor. Jest ona dopasowana do konkretnego typu procesora i wyrażona w postaci rozumianych przez niego kodów rozkazów i ich argumentów. Jest to postać trudna do bezpośredniej analizy przez człowieka, dlatego, by ułatwić sobie zadanie, używa się monitorów kodu maszynowego (program komputerowy służący do nadzoru nad stanem lub czynnościami wykonywanymi przez komputer) lub deasemblerów (program komputerowy, który tłumaczy język maszynowy lub kod bajtowy na język assemblera). Kod maszynowy może być generowany w procesie kompilacji (w przypadku języków wysokiego poziomu) lub asemlacji (w przypadku języków niskiego poziomu). W trakcie procesu generowania kodu maszynowego często tworzony jest przenośny kod pośredni zapisywany w pliku obiektowym. Następnie kod ten pobrany z pliku obiektowego poddawany jest konsolidacji (linkowaniu) z kodem w innych plikach, w celu utworzenia ostatecznej postaci kodu maszynowego, który będzie zapisany w pliku wykonywalnym.

2. Rekurencja a iteracja

Rekurencja jest techniką programowania, dzięki której procedura, funkcja lub podprogram jest w stanie w swoim ciele wywołać sam siebie. Trudno w to uwierzyć, ale niektóre "stare" języki programowania nie udostępniały możliwości wywołań rekurencyjnych. Po co nam jest rekurencja? Przede wszystkim dzięki niej łatwo jest wykonać wiele zadań, w których potrzeba jest wyników częściowych do obliczenia całości. Sztandarowym przykładem w zagadnieniu rekurencji jest liczenie silni ($n!$), lub nieco bardziej zaawansowany przykład liczenia n -tej wartości w ciągu Fibonacciego.

Iteracja (łac. iteratio) to czynność powtarzania (najczęściej wielokrotnego) tej samej instrukcji (albo wielu instrukcji) w pętli. Mianem iteracji określa się także operacje wykonywane wewnątrz takiej pętli. W odróżnieniu od rekurencji, która działa "od góry", iteracja do obliczenia $n+1$ -szej wartości wykorzystuje poprzednią, n -tą iterację. Rekurencja dla obliczenia n -tej wartości potrzebowała zejścia aż do pierwszej wartości. W większości języków programowania istnieje co najmniej kilka instrukcji iteracyjnych. Najważniejsze z nich to instrukcje FOR, WHILE i REPEAT (w języku C DO-WHILE).

3. Algorytm

Jest to skończony ciąg jasno zdefiniowanych czynności koniecznych do wykonania pewnego rodzaju zadań, sposób postępowania prowadzący do rozwiązania problemu.

4. Ciąg Fibonacciego

Ciąg Fibonacciego – ciąg liczb naturalnych określony rekurencyjnie w sposób następujący:

Pierwszy wyraz jest równy 0, drugi jest równy 1, każdy następny jest sumą dwóch poprzednich.

Formalnie:

$$F_n := \begin{cases} 0 & \text{dla } n = 0; \\ 1 & \text{dla } n = 1; \\ F_{n-1} + F_{n-2} & \text{dla } n > 1. \end{cases}$$

Kolejne wyrazy tego ciągu nazywane są liczbami Fibonacciego. Zaliczanie zera do elementów ciągu Fibonacciego zależy od umowy – część autorów definiuje ciąg od $F_1 = F_2 = 1$

Pierwsze dwadzieścia wyrazów ciągu Fibonacciego to:

| F_0 | F_1 | F_2 | F_3 | F_4 | F_5 | F_6 | F_7 | F_8 | F_9 | F_{10} | F_{11} | F_{12} | F_{13} | F_{14} | F_{15} | F_{16} | F_{17} | F_{18} | F_{19} |
|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|
| 0 | 1 | 1 | 2 | 3 | 5 | 8 | 13 | 21 | 34 | 55 | 89 | 144 | 233 | 277 | 610 | 987 | 1597 | 2584 | 4181 |

Obliczanie liczby Fibonacciego

Teoretycznie wartości kolejnych wyrazów ciągu Fibonacciego mogą być obliczone wprost z definicji, jest to jednak metoda na tyle wolna, że stosowanie jej ma tylko sens dla niewielu początkowych wyrazów ciągu, nawet na bardzo szybkich komputerach. Wynika to z tego, że definicja F_n wielokrotnie odwołuje się do wartości poprzednich wyrazów ciągu. Drzewo wywołań takiego algorytmu dla parametru n musi mieć co najmniej F_n liści o wartości 1. Ponieważ ciąg Fibonacciego rośnie wykładniczo, oznacza to wyjątkowo słabą wydajność.

Istnieje równie prosta i znacznie szybsza metoda. Obliczamy wartości ciągu po kolei: F_0 , F_1 , F_2 i tak aż do F_n za każdym razem korzystając z tego, co już obliczyliśmy. Nie trzeba nawet zapamiętywać wszystkich obliczonych dotychczas wartości, ponieważ wystarczą dwie ostatnie. Daje to złożoność liniową – o wiele lepszą od wykładniczej złożoności poprzedniej metody. Metoda ta może być postrzegana jako zastosowanie programowania dynamicznego.

Fibonacci(n)

```
if  $n=0$  then return 0
if  $n=1$  then return 1
 $f' \leftarrow 0$ 
 $f \leftarrow 1$ 
for  $i \leftarrow 2$  to  $n$ 
do
   $m \leftarrow f + f'$ 
   $f' \leftarrow f$ 
   $f \leftarrow m$ 
end
return ( $f$ )
```

5. Tribanacji

6. Przykłady algorytmów

a. Algorytm Euklidesa

Algorytm Euklidesa – algorytm wyznaczania największego wspólnego dzielnika dwóch liczb. Został opisany przez greckiego matematyka, Euklidesa w jego dziele „Elementy”, w księgach siódmej oraz dziesiątej. Najprostsza wersja algorytmu rozpoczyna się od wybrania dwóch liczb naturalnych, dla których należy wyznaczyć największy wspólny dzielnik. Następnie z tych dwóch liczb tworzymy nową parę: pierwszą z liczb jest liczba mniejsza, natomiast drugą jest różnica liczby większej i mniejszej. Proces ten jest powtarzany aż obie liczby będą sobie równe – wartość tych liczb to największy wspólny dzielnik wszystkich par liczb wcześniej

wyznaczonych. Wadą tej wersji algorytmu jest duża liczba operacji odejmowania, które należy wykonać w przypadku, gdy różnica pomiędzy liczbami z pary jest znacząca. Operacja odejmowania mniejszej liczby od większej może zostać zastąpiona przez wyznaczanie reszty z dzielenia. W tej wersji nowa para liczb składa się z mniejszej liczby oraz reszty z dzielenia większej przez mniejszą. Algorytm kończy się w momencie, w którym jedna z liczb jest równa zero – druga jest wtedy największym wspólnym dzielnikiem.

```
int_Euklides (a,b)
    while (b ≠ 0) do
        r ← a mod b
        a ← b
        b ← r}
    return(a)

r_Euklides (a,b)
    if ( b = 0 ) return (a)
    return (r_Euklides (b,a mod (b) ))
```

b. Min-Max

Minimax (czasami minmax) – metoda minimalizowania maksymalnych możliwych strat. Alternatywnie można je traktować jako maksymalizację minimalnego zysku (maximin). Wywodzi się to z teorii gry o sumie zerowej, obejmujących oba przypadki, zarówno ten, gdzie gracze wykonują ruchy naprzemiennie, jak i ten, gdzie wykonują ruchy jednocześnie. Zostało to również rozszerzone na bardziej skomplikowane gry i ogólne podejmowanie decyzji w obecności niepewności.

```
min_max (A, n)
    if ( n mod 2 = 1)
        A[n] ← A[n-1]
    min ← -∞
    max ← ∞
    for( i ← 0; i < n; i+=2)
        if(A[i] > A[i+1])
            if(A[i] > max)
                max ← A[i]
            if(A[i+1] < min)
                min ← A[i+1]
        else
            if(A[i] > min)
                min ← A[i]
            if(A[i+1] < max)
                max ← A[i+1]
    return (min, max)
```

c. Algorytmy sortujące

- **Stabilne**

→ **Bąbelkowe** (ang. bubble sort) – prosta metoda sortowania o złożoności czasowej $O(n^2)$ i pamięciowej $O(1)$. Polega na porównywaniu dwóch kolejnych elementów i zamianie ich kolejności, jeżeli zaburza ona porządek, w jakim się sortuje tablicę. Sortowanie kończy się, gdy podczas kolejnego przejścia nie dokonano żadnej zmiany.

```
BubbleSort( N, B[])
    for(i ← 1; i < N; i++)
        for(j ← 1; j ≤ N-i; j++)
            if(B[j] > B[j+1])
                t ← B[j]
                B[j] ← B[j+1]
                B[j+1] ← t
    return (B[])
```

→ **Przez wstawianie** (ang. Insert Sort, Insertion Sort) – jeden z najprostszych algorytmów sortowania, którego zasada działania odzwierciedla sposób w jaki ludzie ustawiają karty – kolejne elementy wejściowe są ustawiane na odpowiednie miejsca docelowe. Jest efektywny dla niewielkiej liczby elementów, jego złożoność wynosi $O(n^2)$. Pomimo tego, że jest znacznie mniej wydajny od algorytmów takich jak quicksort czy heapsort, posiada pewne zalety:

- liczba wykonanych porównań jest zależna od liczby inwersji w permutacji, dlatego algorytm jest wydajny dla danych wstępnie posortowanych,
- jest wydajny dla zbiorów o niewielkiej liczebności,
- jest stabilny.

Istnieje modyfikacja algorytmu, pozwalająca zmniejszyć liczbę porównań.

Zamiast za każdym razem iterować po już posortowanym fragmencie (etap wstawiania elementu), można posłużyć się wyszukiwaniem binarnym.

Pozwala to zmniejszyć liczbę porównań do $O(n \log n)$, nie zmienia się jednak złożoność algorytmu, ponieważ liczba przesunięć elementów to nadal $O(n^2)$.

```

InsertSort( A[], n){
    for ( i ← 1; i < n; i ++ )
        min ← A[i]
        k ← i
        for( j ← i+1; j ≤ n; j++)
            if( A[j] < min)
                min ← A[j]
                k ← j
        t ← A[i]
        A[i] ← A[k]
        A[k] ← t
    return (A[])

```

→ **Przez scalanie** (ang. merge sort) – rekurencyjny algorytm sortowania danych, stosujący metodę dziel i zwyciężaj. Odkrycie algorytmu przypisuje się Johnowi von Neumannowi.

```

MergeSort(A, left, mid, right)
    i ← left
    j ← mid+1 //mid -> środek
    k ← 1
    while( i ≤ mid & j ≥ right)
        if( A[i] < A[j])
            B[k++] ← A[i++]
        else
            B[k++] ← A[j++]
    while( i ≤ mid)
        B[k++] ← A[i++]
    while( j ≤ right)
        B[k++] ← A[j++]
    i ← left
    j ← 1
    while ( i ≤ mid)
        A[i++] ← B[j++]
    return (A[])

```

→ **Przez zliczanie** (ang. counting sort) – metoda sortowania danych, która polega na sprawdzeniu ile wystąpień kluczy mniejszych od danego występuje w sortowanej tablicy. Algorytm zakłada, że klucze elementów należą do skończonego zbioru (np. są to liczby całkowite z przedziału 0..100), co ogranicza możliwości jego zastosowania.

Algorytm sortowania przez zliczanie zbudowany jest z kolejno następujących po sobie pętli iteracyjnych:

- **W pętli nr 1** przygotowujemy liczniki wystąpień poszczególnych kluczy. Ustawiamy je na 0.
- **W pętli nr 2** przeglądamy kolejne elementy zbioru zwiększając o 1 licznik o numerze równym wartości klucza w sortowanym elemencie zbioru. Po zakończeniu tej pętli w licznikach mamy ilość wystąpień poszczególnych kluczy.
- **W pętli nr 3** przekształcamy zliczone wartości wystąpień kluczy na ostatnie pozycje elementów z danym kluczem w zbiorze wyjściowym.
- **W pętli nr 4** ponownie przeglądamy zbiór wejściowy (idąc od końca do początku, aby zachować kolejność elementów równych - inaczej algorytm nie byłby stabilny) i przesyłamy elementy ze zbioru wejściowego do zbioru wyjściowego na pozycję o numerze zawartym w liczniku skojarzonym z kluczem elementu. Po przesłaniu licznik zmniejszamy o 1, aby kolejny element o tej samej wartości klucza trafił na poprzednią pozycję (idziemy wstecz, zatem kolejność elementów o tym samym kluczu zostanie zachowana). Po zakończeniu tej pętli dane w zbiorze wynikowym są posortowane rosnąco. Kończymy zatem algorytm.

```

CountingSort( A, n, min, max )
  for ( i ← min; i ≤ max; i++ )
    B[i] ← 0
  for ( i ← 1; i ≤ n; i++ )
    B[ A[i].klucz ] ← B[ A[i].klucz ] + 1
  for ( i ← min+1; i ≤ max; i++ )
    B[i] ← B[i] + B[i-1]
  for ( i ← n; i ≥ 1; i-- )
    C[ B[ A[i].klucz ] ] ← A[i]
    B[ A[i].klucz ] ← B[ A[i].klucz ] - 1
  return ( C[] )

```

→ **Kubelkowe** (ang. bucket sort) – jeden z algorytmów sortowania, najczęściej stosowany, gdy liczby w zadanym przedziale są rozłożone jednostajnie, ma on wówczas złożoność $\Theta(n)$. W przypadku ogólnym pesymistyczna złożoność obliczeniowa tego algorytmu wynosi $O(n)$. Pomysł takiego sortowania podali po raz pierwszy w roku 1956 E. J. Issac i R. C. Singleton.

Algorytm realizujemy w czterech pętlach.

- **W pętli nr 1** zerujemy kolejne liczniki $lw[]$.
- **W pętli nr 2** przeglądamy kolejne elementy zbioru od pierwszego do ostatniego. Dla każdego elementu zwiększamy licznik o numerze równym wartości elementu. Po zakończeniu tej pętli liczniki zawierają liczbę wystąpień poszczególnych wartości elementów w sortowanym zbiorze. Zmienna j służy do umieszczania w zbiorze

wyjściowym kolejnych elementów. Umieszczanie rozpoczniemy od początku zbioru, dlatego zmienne ta przyjmuje wartość 1.

- **W pętli nr 3** przeglądamy kolejne liczniki. Jeśli zawartość danego licznika jest większa od zera, to **pętla nr 4** umieści w zbiorze wyjściowym odpowiednią ilość numerów licznika, które odpowiadają zliczonym wartościom elementów ze zbioru wejściowego.

Po zakończeniu pętli nr 3 elementy w zbiorze wyjściowym są posortowane.

```
BucketSort(A, n, min, max)
  for( i ← min; i ≤ max; i++)
    B[i] ← 0
  for(i ← 1; i ≤ max; i++)
    B[ A[i]] ← B[ A[i]] + 1
  j ← 0
  for (i ← min; i ≤ max; i++)
    while(B[i] > 0)
      A[j] ← i
      B[i] ← B[i] – 1
      j++
  return (A[])
```

→ **Pozycyjne** (ang. radix sort) to algorytm sortowania porządkujący stabilnie ciągi wartości (liczb, słów) względem konkretnych cyfr, znaków itp, kolejno od najmniej znaczących do najbardziej znaczących pozycji. Złożoność obliczeniowa jest równa $O(d(n+k))$, gdzie k to liczba różnych cyfr, a d liczba cyfr w kluczach. Wymaga $O(n+k)$ dodatkowej pamięci. Przewagą sortowania pozycyjnego nad innymi metodami jest fakt, iż nie wykonuje ono żadnych operacji porównania na danych wejściowych. Załóżmy że mamy dużą liczbę bardzo długich liczb, bardzo do siebie podobnych – w tym sensie, że większość z nich ma takie same cyfry na początkowych pozycjach. Nie jest łatwo powiedzieć która jest większa, gdyż za każdym razem musimy porównać dużo cyfr zanim trafimy na różnicę. Czas porównania takich liczb jest zatem proporcjonalny do ich długości. Gdybyśmy do posortowania tych liczb zastosowali algorytm porównujący liczby, np. sortowanie szybkie, otrzymalibyśmy dla niego złożoność $O(dn \log n)$ gdzie d to liczba cyfr w liczbach. Algorytmy pozycyjne sprawdzają się także w roli algorytmów sortujących listy.

```

RadixSort(A, n, max)
  for(m ← 1, m ≤ max; m <<=1)
    L0 ← 0
    L1 ← 1
    for( i ← 1; i ≤ n; i++)
      if( A[i] and m ≠ 0)
        L1 ← L1+1
      else L0 ← L0+1
    L1 ← L1 + L0
    for( i ← n; i ≥ 1; i--)
      if(A[i] and m ≠ 0)
        B[L1] ← A[i]
        L1 ← L1 -1
      else
        B[L0] ← A[i]
        L0 ← L0 - 1
    m <<= 1 //przesunięcie m o jeden bit w lewo
    L0 ← 0
    L1 ← 1
    for( i ← 1; i ≤ n; i++)
      if( B[i] and m ≠ 0)
        L1 ← L1+1
      else L0 ← L0+1
    L1 ← L1 + L0
    for( i ← n; i ≥ 1; i--)
      if(B[i] and m ≠ 0)
        A[L1] ← B[i]
        L1 ← L1 -1
      else
        A[L0] ← B[i]
        L0 ← L0 - 1
  return (A[])

```

→ **Biblioteczne** (ang. Library sort) – algorytm sortowania, który bazuje na algorytmie sortowania przez wstawianie, ale z dodawaniem pustych miejsc w tablicy w celu przyspieszenia wstawiania elementów.

- **Niestabilne**

→ **Przez wybieranie** - jedna z prostszych metod sortowania o złożoności **O(n²)**. Polega na wyszukaniu elementu mającego się znaleźć na żądanej pozycji i zamianie miejscami z tym, który jest tam obecnie. Operacja jest wykonywana dla wszystkich indeksów sortowanej tablicy. Algorytm przedstawia się następująco:

- wyszukaj minimalną wartość z tablicy spośród elementów od i do końca tablicy zamień wartość minimalną,
- z elementem na pozycji i

Gdy zamiast wartości minimalnej wybierana będzie maksymalna, wówczas tablica będzie posortowana od największego do najmniejszego elementu.

Algorytm jest niestabilny. Przykładowa lista to: $[2a, 2b, 1] \rightarrow [1, 2b, 2a]$ (gdzie $2b = 2a$)

```
SelectionSort(A, n)
  for( j ← 1; j < n; j++)
    min ← j
    for( i ← j+1; i ≤ n; i++)
      if(A[i] < A[min])
        min ← i
    t ← A[min]
    A[min] ← A[j]
    A[j] ← t
  return (A[])
```

→ **Shella** (ang. Shellsort) – jeden z algorytmów sortowania działających w miejscu i korzystających z porównań elementów. Można go traktować jako uogólnienie sortowania przez wstawianie lub sortowania bąbelkowego, dopuszczające porównania i zamiany elementów położonych daleko od siebie. Na początku sortuje on elementy tablicy położone daleko od siebie, a następnie stopniowo zmniejsza odstęp między sortowanymi elementami. Dzięki temu może je przenieść w docelowe położenie szybciej niż zwykłe sortowanie przez wstawianie. Pierwszą wersję tego algorytmu, której zawdzięcza on swoją nazwę, opublikował w 1959 roku Donald Shell. Złożoność czasowa sortowania Shella w dużej mierze zależy od użytego w nim ciągu odstępów. Wyznaczenie jej dla wielu stosowanych w praktyce wariantów tego algorytmu pozostaje problemem otwartym.

```
ShellSort(A, n)
  for( h ← 1; h ≥ n; h ← 3h + 1)
    h ← h/9
  if(!h) h ← h + 1
  while (h)
    for( j ← n-h-1; j ≥ 0; j--)
      x ← A[j]
      i ← j+h
      while(( i < n) and (x > A[i]))
        A[i-h] ← A[i]
        i ← i+h
      A[i-h] ← x
    h ← h/3
  return (A[])
```

→ **Grzebieniowe**

→ **Szybkie** (ang. quicksort) – jeden z popularnych algorytmów sortowania działających na zasadzie "dziel i zwyciężaj". Sortowanie QuickSort zostało wynalezione w 1962 przez C.A.R. Hoare'a. Algorytm sortowania szybkiego jest wydajny: jego średnia złożoność obliczeniowa jest rzędu **$O(n \log n)$** . Ze względu na szybkość i prostotę implementacji jest powszechnie używany. Jego implementacje znajdują się w bibliotekach standardowych wielu środowisk programowania.

Oznaczenia:

left – indeks pierwszego elementu w nieposortowanej tablicy

right – indeks ostatniego elementu w nieposortowanej tablicy

```
QuickSort(A, left, right)
    i ← (left + right)/2
    piwot ← A[i]
    A[i] ← A[right]
    j ← left
    for ( i ← left; i < right; i++ )
        if ( A[i] < piwot )
            t ← A[i]
            A[i] ← A[j]
            A[j] ← t
            j = j + 1
    A[right] ← A[j]
    A[j] ← piwot
    if ( left < j-1 )
        QuickSort(A, left, j-1)
    if ( j+1 < right )
        QuickSort(A, j+1, right)
    return (A[])
```

→ **Introspektywne**

→ **Przez kopcowanie**

7. Asymptotyczne tempo wzrostu

Asymptotyczne tempo wzrostu jest miarą określającą zachowanie wartości funkcji wraz ze wzrostem jej argumentów. Stosowane jest szczególnie często w teorii obliczeń, w celu opisu złożoności obliczeniowej, czyli zależności ilości potrzebnych zasobów (np. czasu lub pamięci) od rozmiaru danych wejściowych algorytmu. Asymptotyczne tempo wzrostu opisuje jak szybko dana funkcja rośnie lub maleje, abstrahując od konkretnej postaci tych zmian. Do opisu asymptotycznego tempa wzrostu stosuje się notację dużego **O** (omikron) oraz jej modyfikacje, m.in. notacja **Ω** (omega), **Θ** (theta). Notacja dużego **O** została zaproponowana po raz pierwszy w

roku 1894 przez niemieckiego matematyka Paula Bachmanna. W późniejszych latach spopularyzował ją w swoich pracach Edmund Landau, niemiecki matematyk, stąd czasem nazywana jest notacją Landaua.

8. Metody obliczania rekurensji

Czas działania programu

- Dla konkretnych danych wejściowych jest wyrażony liczbą wykonanych prostych (elementarnych) operacji lub “kroków”. Jest dogodne zrobienie założenia że operacja elementarna jest maszynowo niezależna. Θ Każde wykonanie i-tego wiersza programu jest równe ci, przy czym ci jest stałą. Θ
- Kiedy algorytm zawiera rekurencyjne wywołanie samego siebie, jego czas działania można często opisać zależnością rekurencyjną (rekurencja) wyrażającą czas dla problemu rozmiaru n za pomocą czasu dla podproblemów mniejszych rozmiarów. Θ
- Możemy więc użyć narzędzi matematycznych aby rozwiązać rekurencje i w ten sposób otrzymać oszacowania czasu działania algorytmu.

Metody rozwiązywania rekurencji Θ

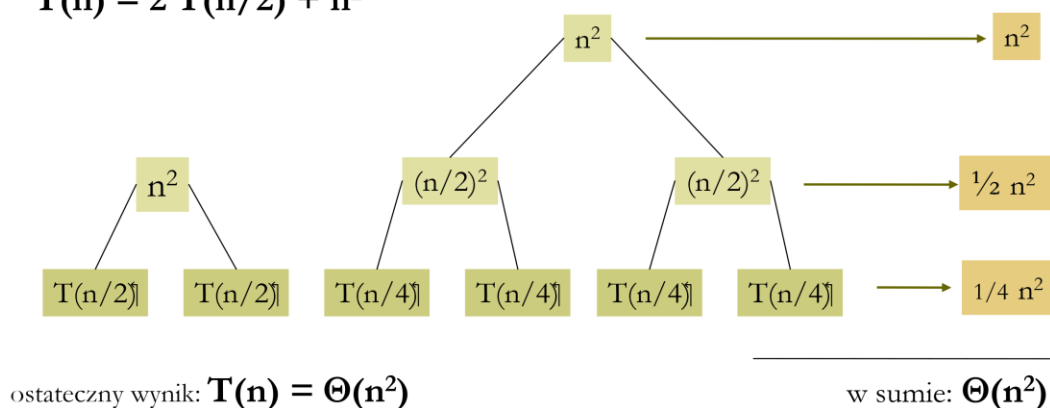
- Metoda podstawiania: Θ zgadujemy oszacowanie, a następnie dowodzimy przez indukcję jego poprawność. Θ
- Metoda iteracyjna: Θ przekształcamy rekurencję na sumę, korzystamy z technik ograniczania sum. Θ
- Metoda uniwersalna: Θ stosujemy oszacowanie na rekurencję mające postać $T(n) = a T(n/b) + f(n)$, gdzie $a \geq 1$, $b > 1$, a $f(n)$ jest daną funkcją.
- Drzewa rekursji

Pozwalają w dogodny sposób zilustrować rozwijanie rekurencji, jak również ułatwiają stosowanie aparatu algebraicznego służącego do rozwiązywania tej rekurencji.

Szczególnie użyteczne gdy rekurencja opisuje algorytm typu “dziel i zwyciężaj”.

Drzewo rekursji dla algorytmu „dziel i zwyciężaj

$$T(n) = 2 T(n/2) + n^2$$



9. Drzewa

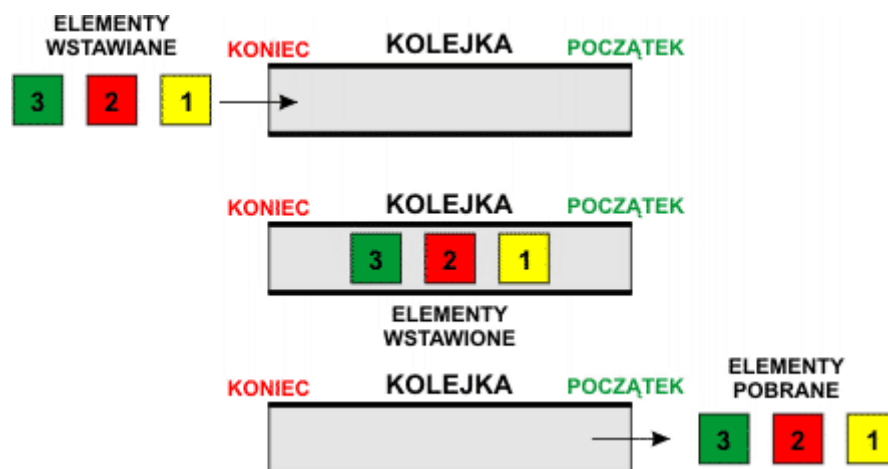
- a. Przechodzenie drzew BFS/DFS
- b. Drzewo poszukiwań binarnych – BST
- c. Kopiec

10. Algorytmy grafowe

- a. Graf
- b. Wzór Eulera
- c. Graf: pełny, zerowy, skierowany i nieskierowany
- d. Macierz sąsiedztwa
- e. Kolejka

Kolejka (ang. queue) jest sekwencyjną strukturą danych, w której dostęp do elementów odbywa się w kolejności ich zapisu, czyli odwrotnie niż dla stosów. Kolejki posiadają mnóstwo zastosowań we współczesnej informatyce, począwszy od prostego buforowania danych, a skończywszy na zaawansowanych algorytmach grafowych. W systemach komputerowych kolejki są wykorzystywane do szeregowania zadań – np. typową kolejką jest kolejka zadań do drukowania, w której są zbierane kolejno dokumenty oczekujące na wydruk na drukarce. Kolejkę możemy

sobie wyobrazić jako tubę – elementy wstawiamy do tuby z jednej strony, po czym przesuują się one wewnątrz i wychodzą z drugiej strony w tej samej kolejności, w jakiej zostały do tuby włożone.



Dla kolejki są zdefiniowane operacje:

- Sprawdzenie, czy kolejka jest pusta – operacja empty zwraca true, jeśli kolejka nie zawiera żadnego elementu, w przeciwnym razie zwraca false.
- Odczyt elementu z początku kolejki – operacja front zwraca wskazanie do elementu, który jest pierwszy w kolejce.
- Zapis elementu na koniec kolejki – operacja push dopisuje nowy element na koniec elementów przechowywanych w kolejce.
- Usunięcie elementu z kolejki – operacja pop usuwa z kolejki pierwszy element.

Jeśli porównasz te operacje z operacjami dostępnymi dla stosu, to okaże się, że obie te struktury są bardzo do siebie podobne. Różnią się jedynie kolejnością dostępu do elementów.

f. Przeszukiwanie wszecz BFS

Zaczynamy odwiedzanie od wierzchołka startowego. Następnie odwiedzamy wszystkich jego sąsiadów. Dalej odwiedzamy wszystkich nieodwiedzonych jeszcze sąsiadów sąsiadów. Itd.

Lista kroków:

Wejście:

v – numer wierzchołka startowego, $v \in C$

visited – wyzerowana tablica logiczna n elementowa z informacją o odwiedzonych wierzchołkach

graf – zadany w dowolnie wybrany sposób, algorytm tego nie precyzuje

Wyjście:

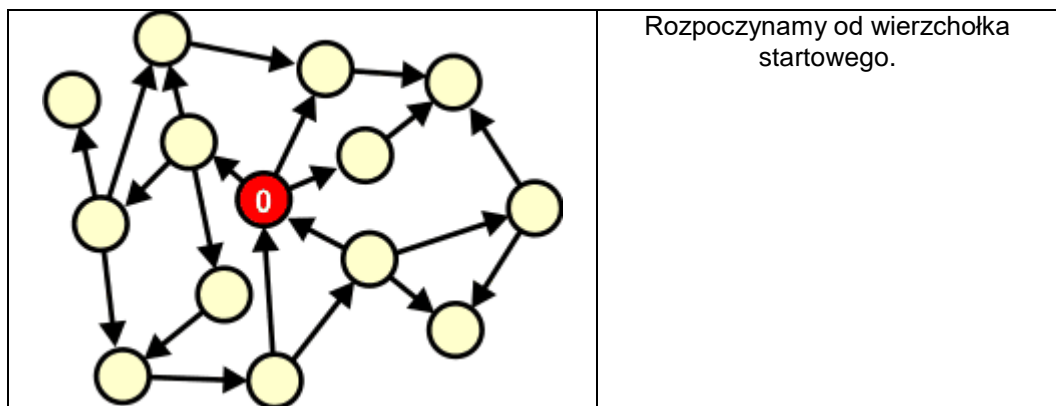
Przetworzenie wszystkich wierzchołków w grafie.

Elementy pomocnicze:

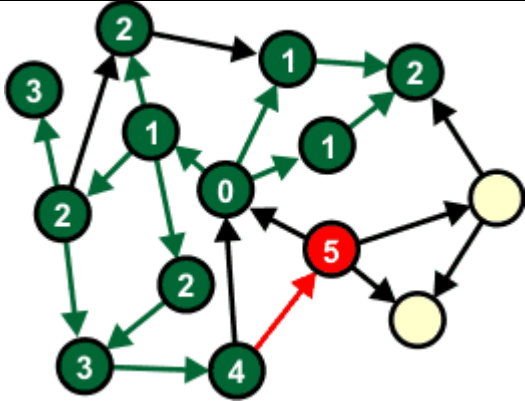
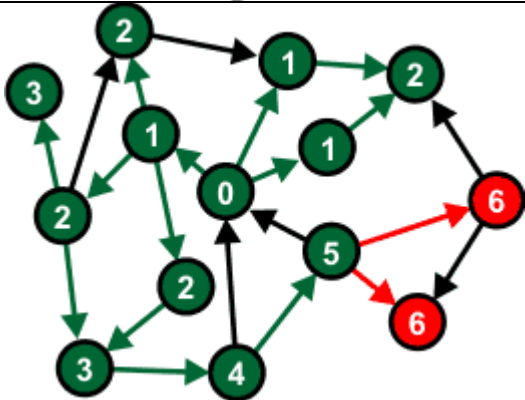
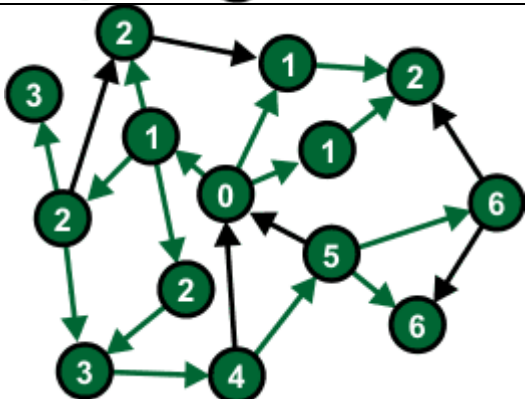
Q – kolejka

u – wierzchołek roboczy, $u \in C$

| | | |
|-----|---|--|
| K01 | Q.push(v) | w kolejce umieszczamy numer wierzchołka startowego |
| K02 | visited[v] \leftarrow true | oznaczamy wierzchołek jako odwiedzony |
| K03 | Dopóki Q.empty() = false, wykonuj K04...K10 | tutaj jest pętla główna algorytmu BFS |
| K04 | v \leftarrow Q.front() | odczytujemy z kolejki numer wierzchołka |
| K05 | Q.pop() | odczytany numer usuwamy z kolejki |
| K06 | Przetwórz wierzchołek v | tutaj wykonujemy operacje na wierzchołku v |
| K07 | Dla każdego sąsiada u wierzchołka v wykonuj K08...K10 | przeglądamy wszystkich sąsiadów v |
| K08 | Jeśli visited[u] = false, to następny obieg pętli K07 | szukamy nieodwiedzonego sąsiada |
| K09 | Q.push(u) | umieszczamy go w kolejce |
| K10 | visited[u] \leftarrow true | i oznaczamy jako odwiedzony |
| K11 | Zakończ | |

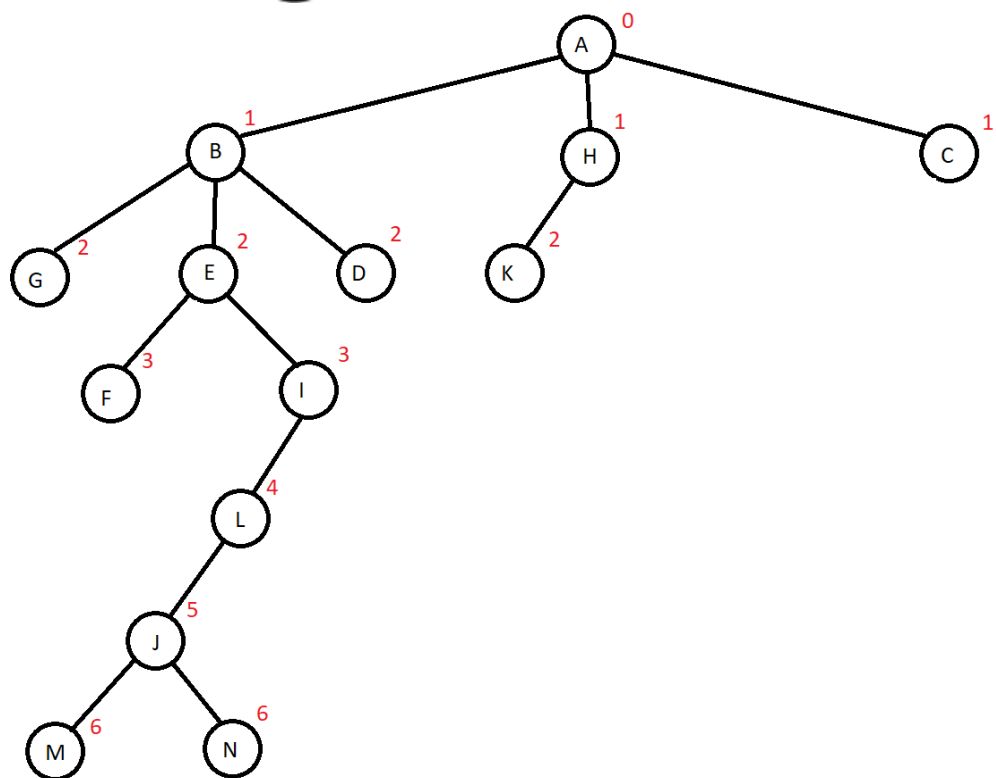
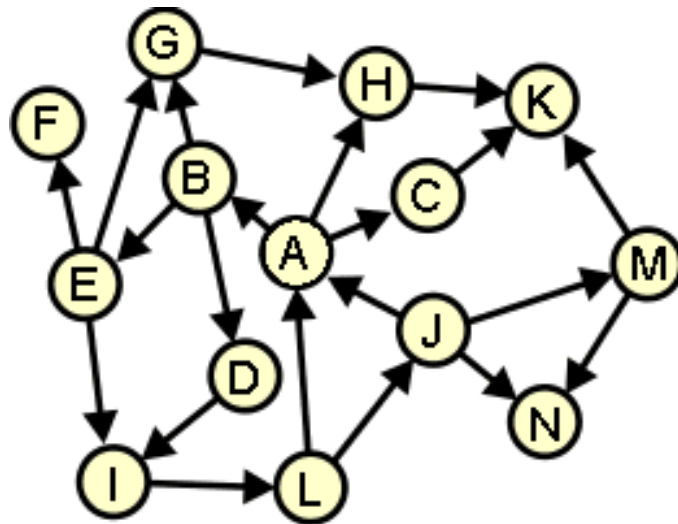
Przykład grafu:

| | |
|--|---|
| | <p>Wierzchołek startowy zaznaczamy jako odwiedzony (zielony) i odwiedzamy wszystkich jego sąsiadów pierwszego poziomu.</p> |
| | <p>Sąsiadów pierwszego poziomu oznaczamy jako odwiedzonych i odwiedzamy wszystkich ich sąsiadów, którzy jeszcze nie byli odwiedzeni.</p> |
| | <p>Sąsiadów drugiego poziomu oznaczamy jako odwiedzonych i odwiedzamy wszystkich ich sąsiadów, którzy jeszcze nie byli odwiedzeni.</p> |
| | <p>Sąsiadów trzeciego poziomu oznaczamy jako odwiedzonych i odwiedzamy wszystkich ich sąsiadów, którzy jeszcze nie byli odwiedzeni. Do odwiedzenia jest teraz tylko jeden taki wierzchołek.</p> |

| | |
|--|--|
|  | <p>Sąsiada czwartego poziomu oznaczamy jako odwiedzonego i odwiedzamy wszystkich jego sąsiadów, którzy jeszcze nie byli odwiedzeni. Do odwiedzenia jest znów tylko jeden taki wierzchołek.</p> |
|  | <p>Sąsiada piątego poziomu oznaczamy jako odwiedzonego i odwiedzamy wszystkich jego sąsiadów, którzy jeszcze nie byli odwiedzeni. Do odwiedzenia pozostały dwa ostatnie wierzchołki.</p> |
|  | <p>Oznaczamy jako odwiedzonych dwóch sąsiadów stopnia 6. W grafie nie ma już nieodwiedzonych wierzchołków. Przejście zostało zakończone.</p> |

Taki sposób odwiedzania wierzchołków wymaga kolejki. Powód jest bardzo prosty. Kolejka jest sekwencyjną strukturą danych, z której odczytujemy elementy w kolejności ich zapisu. Na koniec kolejki wstawiamy wierzchołek startowy. Wewnątrz pętli wierzchołek ten zostanie odczytany z początku kolejki, po czym algorytm umieści w niej wszystkich nieodwiedzonych sąsiadów. W kolejnych obiegach pętli sąsiedzi ci (sąsiedzi poziomu 1) zostaną odczytani z początku kolejki, a na jej koniec zostaną wstawieni sąsiedzi poziomu 2. Gdy wszyscy sąsiedzi poziomu 1 zostaną przetworzeni, w kolejce pozostaną tylko sąsiedzi poziomu 2. Teraz oni będą odczytywani z początku kolejki, a na jej koniec trafią sąsiedzi poziomu 3. Całość będzie się powtarzała w pętli dotąd, aż algorytm przetworzy wszystkie dostępne wierzchołki w grafie.

Graf ten możemy też przedstawić w formie drzewa przypisując wierzchołkom wartości np. liczby lub litery np.:



g. Przeszukiwanie w głąb DFS

Zasada działania **DFS** jest następująca:

Zaznaczamy bieżący wierzchołek jako odwiedzony. Przechodzimy do kolejnych sąsiadów wierzchołka bieżącego i wykonujemy dla nich tę samą operację (tzn. zaznaczamy je jako odwiedzony i przechodzimy do ich sąsiadów). Przechodzenie kończymy, gdy zostaną w ten sposób odwiedzony wszystkie dostępne wierzchołki.

Algorytm DFS występuje w dwóch postaciach: rekurencyjnej i stosowej.

Implementacja algorytmu zależy od wybranej reprezentacji grafu.

Lista kroków dla rekurencyjnego DFS:

Wejście:

v – numer wierzchołka startowego, $v \in C$

visited – tablica logiczna n elementowa z informacją o odwiedzonych wierzchołkach

graf – zadany w dowolnie wybrany sposób, algorytm tego nie precyzuje

Wyjście:

Przetworzenie wszystkich wierzchołków w grafie.

Elementy pomocnicze:

u – wierzchołek roboczy, $w \in C$

| | | |
|-----|---|--|
| K01 | visited[v] \leftarrow true | odwiedź wierzchołek |
| K02 | Przetwórz wierzchołek v | przetwarzanie wstępne |
| K03 | Dla każdego sąsiada u wierzchołka v wykonaj: Jeśli visited[u] = false, to DFS(u) | odwiedź algorytmem DFS każdego nieodwiedzonego sąsiada |
| K04 | Przetwórz wierzchołek v | przetwarzanie końcowe |
| K05 | Zakończ | |

Lista kroków dla stosownego algorytmu DFS:

Algorytm stosowy DFS wykorzystuje stos do przechowywania numerów wierzchołków do odwiedzenia.

Wejście:

v – numer wierzchołka startowego, $v \in C$

visited – wyzerowana tablica logiczna o n elementach

graf – zadany w dowolnie wybrany sposób, algorytm tego nie precyzuje

Wyjście:

Przetworzenie wszystkich wierzchołków w grafie.

Elementy pomocnicze:

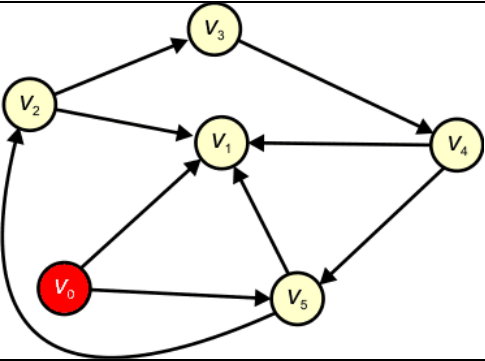
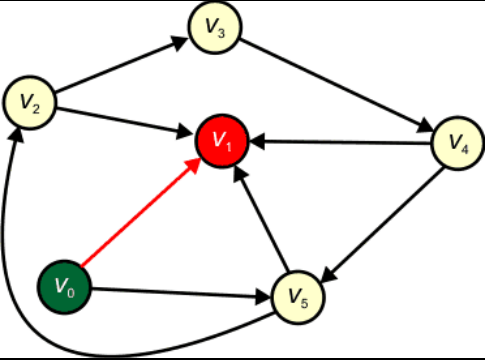
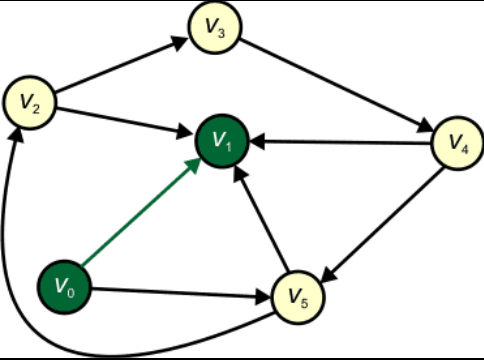
S – pusty stos liczb całkowitych

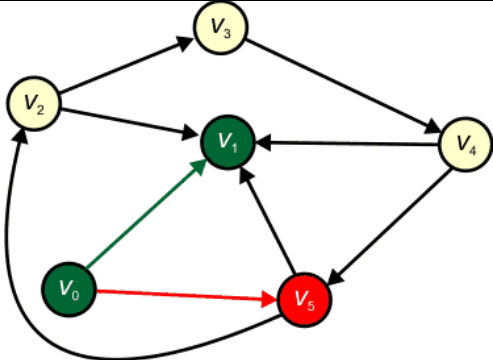
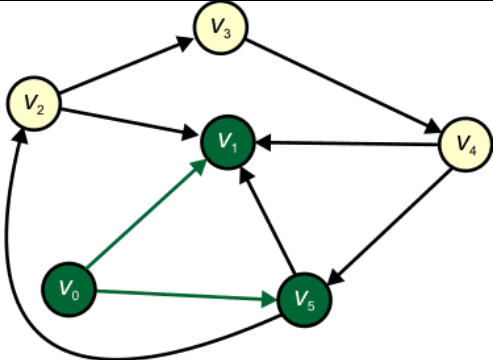
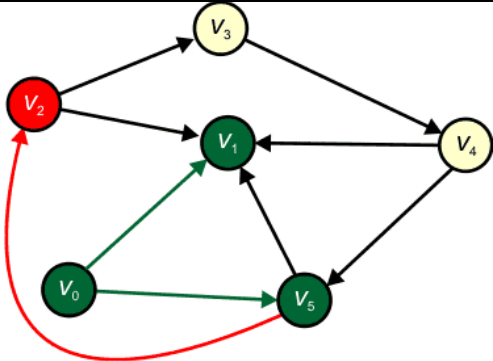
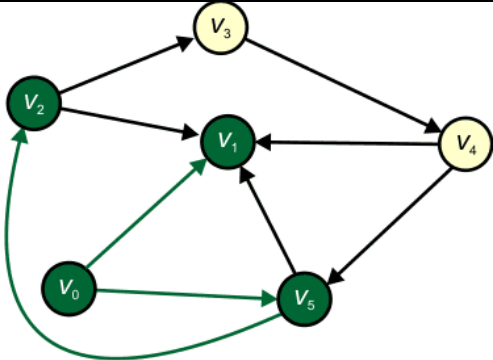
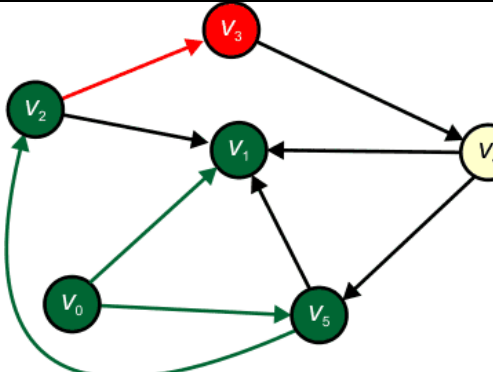
u – wierzchołek roboczy, $u \in C$

| | | |
|-----|---|--|
| K01 | $S.push(v)$ | umieszczamy na stosie numer wierzchołka startowego |
| K02 | visited[v] \leftarrow true | wierzchołek oznaczamy jako odwiedzony |
| K03 | Dopóki $S.empty() = \text{false}$, wykonuj K04...K10 | |
| K04 | $v \leftarrow S.top()$ | pobieramy ze stosu numer wierzchołka |
| K05 | $S.pop()$ | numer usuwamy ze stosu |
| K06 | Przetwórz wierzchołek v | |

| | | |
|-----|---|--|
| K07 | Dla każdego sąsiada u wierzchołka v wykonuj K08...K10 | na stos przesyłamy numery nieodwiedzonych jeszcze wierzchołków |
| K08 | Jeśli $visited[u] = true$, to następny obieg pętli K07 | pomijamy odwiedzonych sąsiadów |
| K09 | $S.push(u);$ | sąsiada umieszczamy na stosie |
| K10 | $visited[u] \leftarrow true$ | oznaczamy go jako odwiedzony |
| K11 | Zakończ | |

Przykład grafu:

| | |
|---|---|
|  | Przejście rozpoczynamy od wierzchołka v_0 |
|  | Wierzchołek v_0 oznaczamy jako odwiedzony (kolor zielony) i przechodzimy do jego sąsiada v_1 |
|  | Wierzchołek v_1 oznaczamy jako odwiedzony. Ponieważ nie ma on sąsiadów, to ta gałąź przejścia jest ślepa i już dalej nie biegnie. |

| | |
|---|---|
|  | <p>Przechodzimy do kolejnego sąsiada wierzchołka v_0, czyli do v_5.</p> |
|  | <p>Wierzchołek v_5 oznaczamy jako odwiedzony. Wierzchołek ten posiada dwóch sąsiadów: v_1 i v_2. Do v_1 nie będziemy przechodzić, ponieważ jest on oznaczony jako już odwiedzony.</p> |
|  | <p>Przechodzimy do v_2.</p> |
|  | <p>Wierzchołek v_2 oznaczamy jako odwiedzony. Posiada on dwóch sąsiadów: v_1 (już odwiedzony) oraz v_3.</p> |
|  | <p>Przechodzimy do v_3.</p> |

| | |
|--|--|
| | <p>Zaznaczamy v_3 jako odwiedzone. Wierzchołek v_3 posiada tylko jednego sąsiada: v_4</p> |
| | <p>Przechodzimy do v_4.</p> |
| | <p>Zaznaczamy v_4 jako odwiedzone. Wierzchołek v_4 posiada dwóch sąsiadów, v_1 i v_5, lecz oba te wierzchołki są już odwiedzone. Przejście kończymy, ponieważ w grafie nie ma już dalszych wierzchołków, do których moglibyśmy przejść. Kolejno odwiedzone wierzchołki: $v_0 \ v_1 \ v_5 \ v_2 \ v_3 \ v_4$</p> |

h. Algorytm Bellmana-Forda

Jeśli ścieżki grafu posiadają nieujemne wagi, to najlepszym rozwiązaniem tego problemu jest algorytm Dijkstry. W niektórych zastosowaniach ścieżki mogą posiadać wagi ujemne. W takim przypadku musimy użyć nieco mniej efektywnego, lecz bardziej wszechstronnego algorytmu Bellmana-Forda. Algorytm tworzy poprawny wynik tylko wtedy, gdy graf nie zawiera ujemnego cyklu (ang. negative cycle), czyli cyklu, w którym suma wag krawędzi jest ujemna. Jeśli taki cykl istnieje w grafie, to każdą ścieżkę można "skrócić" przechodząc wielokrotnie przez cykl ujemny. W takim przypadku algorytm Bellmana-Forda zgłasza błąd.

Opisany tutaj algorytm będzie tworzył dwie n elementowe tablice danych (n oznacza liczbę wierzchołków w grafie):

- d – element i -ty zawiera koszt dojścia z wierzchołka startowego do i -tego wierzchołka grafu po najkrótszej ścieżce. Dla wierzchołka startowego koszt dojścia wynosi 0.
- p – element i -ty zawiera numer wierzchołka grafu, który jest poprzednikiem wierzchołka i -tego na najkrótszej ścieżce. Dla wierzchołka startowego poprzednikiem jest -1.

Na początku algorytmu ustawiamy wszystkie komórki tablicy d na największą możliwą wartość (oryginalnie na nieskończoność) za wyjątkiem komórki odwzorowującej wierzchołek startowy, w której umieszczamy 0. Natomiast we wszystkich komórkach tablicy p umieszczamy -1 (w grafie nie ma wierzchołka o numerze -1, oznacza to zatem brak poprzednika).

Następnie wykonujemy $n - 1$ obiegów pętli, w której dokonujemy relaksacji krawędzi (każdy obieg ustala koszt dojścia do przynajmniej jednego wierzchołka grafu, ponieważ wierzchołek startowy ma koszt 0, to pozostaje nam ustalenie kosztu jeszcze dla $n - 1$ wierzchołków, stąd wymagane jest co najwyżej $n - 1$ obiegów pętli). Polega ona na tym, iż przeglądamy po kolei wszystkie krawędzie grafu. Jeśli natrafimy na krawędź $u-v$ o wadze w , dla której koszt dojścia $d[v]$ jest większy od kosztu dojścia $d[u] + w$ (czyli dojście do wierzchołka v od wierzchołka u tą krawędzią jest tańsze od poprzednio znalezionych dojść), to ustawiamy koszt $d[v]$ na $d[u] + w$ i w tablicy poprzedników dla $p[v]$ umieszczamy numer wierzchołka u . Gdy pętla wykona $n - 1$ obiegów, w tablicy d będziemy mieli koszty dojść do poszczególnych wierzchołków grafu po najkrótszych ścieżkach, a w tablicy p dla każdego wierzchołka znajdziemy jego poprzednik na najkrótszej ścieżce od wierzchołka startowego.

Należy jeszcze sprawdzić, czy w grafie nie występuje cykl ujemny. W tym celu jeszcze raz przeglądamy zbiór krawędzi i jeśli natrafimy na krawędź $u-v$ o wadze w dla której dalej koszt dojścia $d[v]$ jest większy od $d[u] + w$, to mamy do czynienia z cyklem ujemnym (normalnie sytuacja taka nie może wystąpić, ponieważ relaksacja powinna ustawić $d[v]$ na $d[u] + w$. Tylko w przypadku istnienia cyklu ujemnego relaksacja sobie z tym nie radzi). W takim przypadku algorytm powinien zgłosić błąd.

Lista kroków:

Wejście:

n – liczba wierzchołków w grafie, $n \in \mathbb{C}$

graf – zadany w dowolnie wybrany sposób, algorytm tego nie precyzuje. Graf musi być spójny.

v – numer wierzchołka startowego, $v \in \mathbb{C}$

Wyjście:

- true:
 d – n elementowa tablica z kosztami dojścia. Element $d[i]$ zawiera

koszt najkrótszej ścieżki od wierzchołka v do i .

p – n elementowa tablica poprzedników. Element $p[i]$ zawiera numer wierzchołka, który jest poprzednikiem wierzchołka i -tego na najkrótszej ścieżce od wierzchołka v do i . Element $p[v] = -1$, ponieważ wierzchołek startowy nie posiada poprzednika.

- false:
graf zawiera ujemny cykl, który uniemożliwia wyznaczenie najkrótszych ścieżek (każda ścieżka może być najkrótszą, jeśli przepuścimy ją odpowiednią liczbę razy przez cykl ujemny).

Elementy pomocnicze:

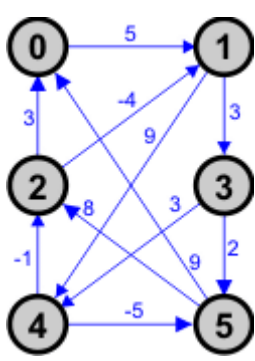
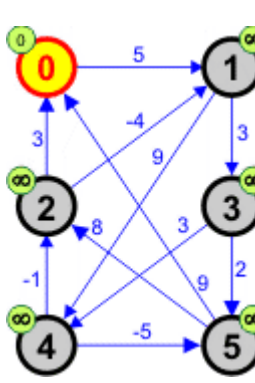
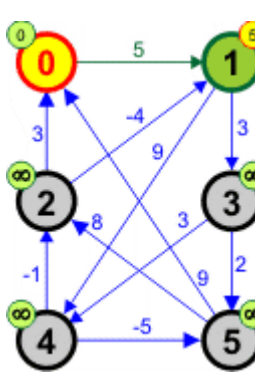
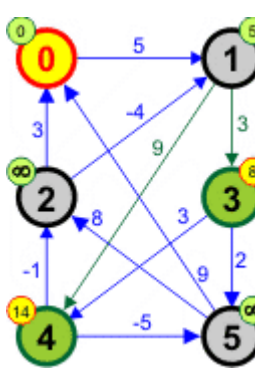
x, y – numery wierzchołków w grafie, $x, y \in C$

i – licznik pętli, $i \in C$

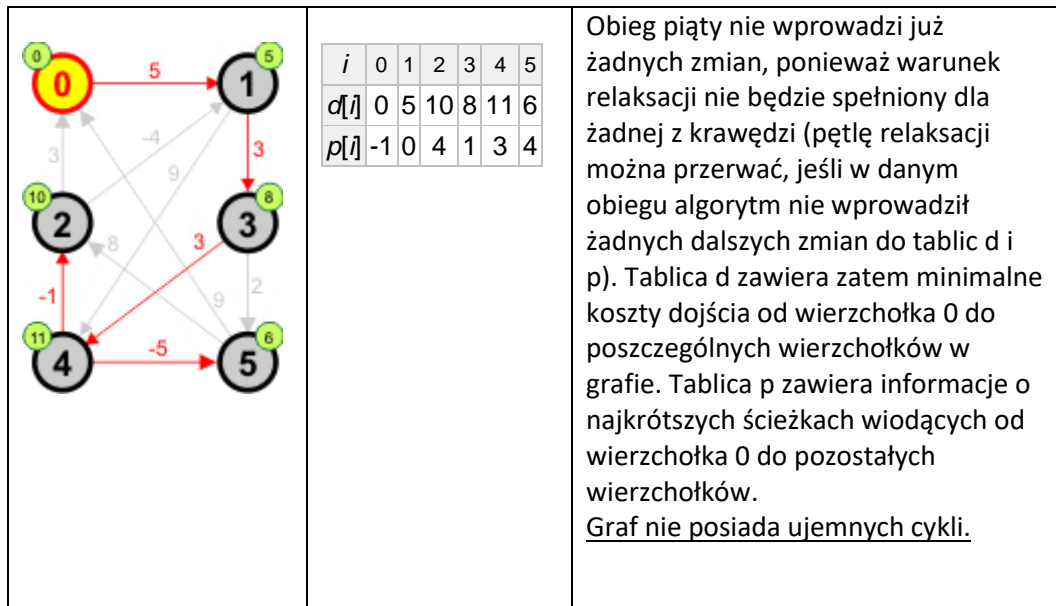
test – logiczna zmienna decyzyjna

| | | |
|-----|--|--|
| K01 | Utwórz n elementową tablicę d i wypełnij ją największą liczbą | |
| K02 | Utwórz n elementową tablicę p i wypełnij ją liczbami -1 | |
| K03 | $d[v] \leftarrow 0$ | koszt dojścia do wierzchołka startowego |
| K04 | Dla $i = 2, 3, \dots, n$ wykonuj K05...K12 | pętlę główną wykonujemy co najwyżej $n - 1$ razy |
| K05 | test \leftarrow true | zmienna przechowuje informację o zmianach |
| K06 | Dla $x = 0, 1, \dots, n-1$ wykonuj K07...K11 | |
| K07 | Dla każdego sąsiada y wierzchołka x wykonuj K08...K11 | |
| K08 | Jeśli $d[y] \leq d[x] + \text{waga krawędzi } x-y$, to następny obieg pętli K07 | sprawdzamy warunek relaksacji krawędzi |
| K09 | test \leftarrow false | zapamiętujemy zmianę |
| K10 | $d[y] \leftarrow d[x] + \text{waga krawędzi } x-y$ | dokonujemy relaksacji krawędzi |
| K11 | $p[y] \leftarrow x$ | ustawiamy poprzednik wierzchołka y na x |
| K12 | Jeśli test = true, to zakończ z wynikiem true | wynik w d i p |
| K13 | Dla $x = 0, 1, \dots, n - 1$ wykonuj K14 | sprawdzamy istnienie ujemnego cyklu |
| K14 | Dla każdego sąsiada y wierzchołka x wykonuj Jeśli $d[y] > d[x] + \text{waga krawędzi } x-y$, to zakończ z wynikiem false | ujemny cykl!!! |
| K15 | Zakończ z wynikiem true | |

Prześledźmy na przykładzie sposób pracy algorytmu Bellmana-Forda:

| | | | | | | | | | | | | | | | | | | | | | | | |
|---|--|--|----------|----------|----------|----------|---|---|--------|---|----------|----------|----------|----------|----------|--------|----|----|----|----|----|----|---|
|  | | Oto nasz graf ważony z ujemnymi wagami krawędzi. | | | | | | | | | | | | | | | | | | | | | |
|  | <table border="1" data-bbox="676 703 911 826"><tr><td>i</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr><tr><td>$d[i]$</td><td>0</td><td>∞</td><td>∞</td><td>∞</td><td>∞</td><td>∞</td></tr><tr><td>$p[i]$</td><td>-1</td><td>-1</td><td>-1</td><td>-1</td><td>-1</td><td>-1</td></tr></table> | i | 0 | 1 | 2 | 3 | 4 | 5 | $d[i]$ | 0 | ∞ | ∞ | ∞ | ∞ | ∞ | $p[i]$ | -1 | -1 | -1 | -1 | -1 | -1 | Wybieramy na wierzchołek startowy wierzchołek o numerze zero. Policzmy koszty dojść do pozostałych wierzchołków po najkrótszych ścieżkach. Tworzymy tablice d i p , wypełniając je odpowiednio |
| i | 0 | 1 | 2 | 3 | 4 | 5 | | | | | | | | | | | | | | | | | |
| $d[i]$ | 0 | ∞ | ∞ | ∞ | ∞ | ∞ | | | | | | | | | | | | | | | | | |
| $p[i]$ | -1 | -1 | -1 | -1 | -1 | -1 | | | | | | | | | | | | | | | | | |
|  | <table border="1" data-bbox="676 1126 911 1252"><tr><td>i</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr><tr><td>$d[i]$</td><td>0</td><td>5</td><td>∞</td><td>∞</td><td>∞</td><td>∞</td></tr><tr><td>$p[i]$</td><td>-1</td><td>0</td><td>-1</td><td>-1</td><td>-1</td><td>-1</td></tr></table> | i | 0 | 1 | 2 | 3 | 4 | 5 | $d[i]$ | 0 | 5 | ∞ | ∞ | ∞ | ∞ | $p[i]$ | -1 | 0 | -1 | -1 | -1 | -1 | W pierwszym obiegu relaksujemy krawędź 0–1, dla której mamy: $d[0] = 0$ $d[1] = \infty$ $d[1] > d[0] + 5$ Ustawiamy: $d[1] \leftarrow 0 + 5 = 5$ $p[1] \leftarrow 0$ (do wierzchołka 1 przychodzimy z wierzchołka 0). |
| i | 0 | 1 | 2 | 3 | 4 | 5 | | | | | | | | | | | | | | | | | |
| $d[i]$ | 0 | 5 | ∞ | ∞ | ∞ | ∞ | | | | | | | | | | | | | | | | | |
| $p[i]$ | -1 | 0 | -1 | -1 | -1 | -1 | | | | | | | | | | | | | | | | | |
|  | <table border="1" data-bbox="676 1545 911 1668"><tr><td>i</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr><tr><td>$d[i]$</td><td>0</td><td>5</td><td>∞</td><td>8</td><td>14</td><td>∞</td></tr><tr><td>$p[i]$</td><td>-1</td><td>0</td><td>-1</td><td>1</td><td>1</td><td>-1</td></tr></table> | i | 0 | 1 | 2 | 3 | 4 | 5 | $d[i]$ | 0 | 5 | ∞ | 8 | 14 | ∞ | $p[i]$ | -1 | 0 | -1 | 1 | 1 | -1 | W drugim obiegu relaksujemy krawędzie 1–3 i 1–4. $d[1] = 5$ $d[3] = \infty$ $d[3] > d[1] + 3$ $d[3] \leftarrow 5 + 3 = 8$ $p[3] \leftarrow 1$ $d[1] = 5$ $d[4] = \infty$ $d[4] > d[1] + 9$ $d[4] \leftarrow 5 + 9 = 14$ $p[4] \leftarrow 1$ |
| i | 0 | 1 | 2 | 3 | 4 | 5 | | | | | | | | | | | | | | | | | |
| $d[i]$ | 0 | 5 | ∞ | 8 | 14 | ∞ | | | | | | | | | | | | | | | | | |
| $p[i]$ | -1 | 0 | -1 | 1 | 1 | -1 | | | | | | | | | | | | | | | | | |

| | <table><tr><th>i</th><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr><tr><th>$d[i]$</th><td>0</td><td>5</td><td>13</td><td>8</td><td>11</td><td>9</td></tr><tr><th>$p[i]$</th><td>-1</td><td>0</td><td>4</td><td>1</td><td>3</td><td>4</td></tr></table> | i | 0 | 1 | 2 | 3 | 4 | 5 | $d[i]$ | 0 | 5 | 13 | 8 | 11 | 9 | $p[i]$ | -1 | 0 | 4 | 1 | 3 | 4 | <p>W trzecim obiegu relaksujemy krawędzie 4–2, 4–5 i 3–4 (zakładamy najbardziej niekorzystną kolejność relaksacji).</p> <p>$d[4] = 14$ $d[2] = \infty$ $d[2] > d[4] + -1$ $d[2] \leftarrow 14 + -1 = 13$ $p[2] \leftarrow 4$</p> <p>$d[4] = 14$ $d[5] = \infty$ $d[5] > d[4] + -5$ $d[5] \leftarrow 14 + -5 = 9$ $p[5] \leftarrow 4$</p> <p>$d[3] = 8$ $d[4] = 14$ $d[4] > d[3] + 3$ $d[4] \leftarrow 8 + 3 = 11$ $p[4] \leftarrow 3$ (zmiana!)</p> |
|--------|--|-----|----|---|----|---|---|---|--------|---|---|----|---|----|---|--------|----|---|---|---|---|---|---|
| i | 0 | 1 | 2 | 3 | 4 | 5 | | | | | | | | | | | | | | | | | |
| $d[i]$ | 0 | 5 | 13 | 8 | 11 | 9 | | | | | | | | | | | | | | | | | |
| $p[i]$ | -1 | 0 | 4 | 1 | 3 | 4 | | | | | | | | | | | | | | | | | |
| | <table><tr><th>i</th><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr><tr><th>$d[i]$</th><td>0</td><td>5</td><td>10</td><td>8</td><td>11</td><td>6</td></tr><tr><th>$p[i]$</th><td>-1</td><td>0</td><td>4</td><td>1</td><td>3</td><td>4</td></tr></table> | i | 0 | 1 | 2 | 3 | 4 | 5 | $d[i]$ | 0 | 5 | 10 | 8 | 11 | 6 | $p[i]$ | -1 | 0 | 4 | 1 | 3 | 4 | <p>W czwartym obiegu relaksujemy krawędzie 4–2 i 4–5</p> <p>$d[4] = 11$ $d[2] = 13$ $d[2] > d[4] + -1$ $d[2] \leftarrow 11 + -1 = 10$ $p[2] \leftarrow 4$</p> <p>$d[4] = 11$ $d[5] = 9$ $d[5] > d[4] + -5$ $d[5] \leftarrow 11 + -5 = 6$ $p[5] \leftarrow 4$</p> |
| i | 0 | 1 | 2 | 3 | 4 | 5 | | | | | | | | | | | | | | | | | |
| $d[i]$ | 0 | 5 | 10 | 8 | 11 | 6 | | | | | | | | | | | | | | | | | |
| $p[i]$ | -1 | 0 | 4 | 1 | 3 | 4 | | | | | | | | | | | | | | | | | |



i. Algorytm DIJKSTRY

Jeśli wagi krawędzi są nieujemne, to problem znalezienia ścieżki o najniższym koszcie dojścia elegancko rozwiązuje algorytm Dijkstry. Algorytm ten pozwala znaleźć koszty dojścia od wierzchołka startowego v do każdego innego wierzchołka w grafie (o ile istnieje odpowiednia ścieżka). Dodatkowo wyznacza on poszczególne ścieżki. Zasada pracy jest następująca:

Tworzymy dwa zbiory wierzchołków Q i S . Początkowo zbiór Q zawiera wszystkie wierzchołki grafu, a zbiór S jest pusty. Dla wszystkich wierzchołków u grafu za wyjątkiem startowego v ustawiamy koszt dojścia $d(u)$ na nieskończoność. Koszt dojścia $d(v)$ zerujemy. Dodatkowo ustawiamy poprzednik $p(u)$ każdego wierzchołka u grafu na niezdefiniowany. Poprzedniki będą wyznaczały w kierunku odwrotnym najkrótsze ścieżki od wierzchołków u do wierzchołka startowego v . Teraz w pętli dopóki zbiór Q zawiera wierzchołki, wykonujemy następujące czynności:

1. Wybieramy ze zbioru Q wierzchołek u o najmniejszym koszcie dojścia $d(u)$.
2. Wybrany wierzchołek u usuwamy ze zbioru Q i dodajemy do zbioru S .
3. Dla każdego sąsiada w wierzchołka u , który jest wciąż w zbiorze Q , sprawdzamy, czy $d(w) > d(u) + \text{waga krawędzi } u-w$. Jeśli tak, to wyznaczamy nowy koszt dojścia do wierzchołka w jako: $d(w) \leftarrow d(u) + \text{waga krawędzi } u-w$. Następnie wierzchołek u czynimy poprzednikiem w : $p(w) \leftarrow u$.

Lista kroków:**Wejście:**

n – liczba wierzchołków w grafie, $n \in \mathbb{C}$

graf – zadany w dowolnie wybrany sposób, algorytm tego nie precyzuje. Definicja grafu powinna udostępniać wagi krawędzi.

v – wierzchołek startowy, $v \in \mathbb{C}$

Wyjście:

d – n elementowa tablica z kosztami dojścia od wierzchołka v do wierzchołka i -tego wzdłuż najkrótszej ścieżki. Koszt dojścia jest sumą wag krawędzi, przez które przechodzimy posuwając się wzdłuż wyznaczonej najkrótszej ścieżki.

p – n elementowa tablica z poprzednikami wierzchołków na wyznaczonej najkrótszej ścieżce. Dla i -tego wierzchołka grafu $p[i]$ zawiera numer wierzchołka poprzedzającego na najkrótszej ścieżce

Elementy pomocnicze:

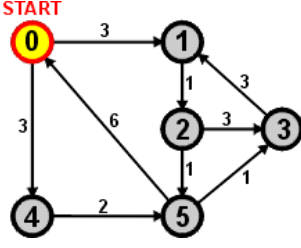
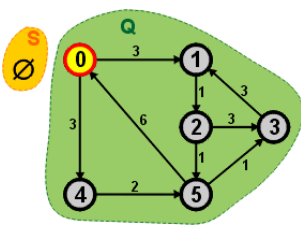
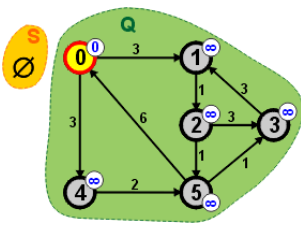
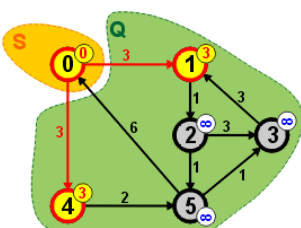
S – zbiór wierzchołków grafu o policzonych już najkrótszych ścieżkach od wybranego wierzchołka v

Q – zbiór wierzchołków grafu, dla których najkrótsze ścieżki nie zostały jeszcze policzone

u, w – wierzchołki, $u, w \in \mathbb{C}$

| | | |
|------|---|---|
| K01 | $S \leftarrow \emptyset$ | zbiór S ustawiamy jako pusty |
| K02 | $Q \leftarrow$ wszystkie wierzchołki grafu | |
| K03 | Utwórz n elementową tablicę d | tablica na koszty dojścia |
| K04 | Utwórz n elementową tablicę p | tablica poprzedników na ścieżkach |
| K05 | Tablicę d wypełnij największą wartością dodatnią | |
| K06 | $d[v] \leftarrow 0$ | koszt dojścia do samego siebie jest zawsze zerowy |
| K07 | Tablicę p wypełnij wartościami -1 | -1 oznacza brak poprzednika |
| K08 | Dopóki Q zawiera wierzchołki, wykonuj K09...K12 | |
| K09 | Z Q do S przenieś wierzchołek u o najmniejszym $d[u]$ | |
| K10 | Dla każdego sąsiada w wierzchołka u , wykonuj K11...K12 | przeglądamy sąsiadów przeniesionego wierzchołka |
| K11 | Jeśli w nie jest w Q , to następny obieg pętli K10 | szukamy sąsiadów obecnych w Q |
| K12 | Jeśli $d[w] > d[u] + \text{waga krawędzi } u-w$, to | sprawdzamy koszt dojścia |
| K12a | $d[w] \leftarrow d[u] + \text{waga krawędzi } u-w$ | jeśli mamy niższy, to modyfikujemy koszt |
| K12b | $p[w] \leftarrow u$ | i zmieniamy poprzednika w na u |
| K13 | Zakończ | |

Aby lepiej zrozumieć zasadę działania algorytmu Dijkstry, prześledźmy jego kolejne kroki w poniższej tabelce:

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|--|---|--|----------|----------|----------|----------|---|---|--------|---|----------|----------|----------|----------|----------|--------|---|---|---|---|---|---|--|---|---|---|---|---|---|--|
| <div><div>START</div></div> | | <p>Mamy ważony graf skierowany z wierzchołkiem startowym $v = 0$. Będziemy wyznaczać najniższe koszty dojścia od wyróżnionego wierzchołka do wszystkich pozostałych wierzchołków w grafie oraz najkrótsze ścieżki pomiędzy wyróżnionym wierzchołkiem, a wszystkimi pozostałymi.</p> | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| <div></div> | | <p>Tworzymy dwa zbiory S i Q. Zbiór S jest początkowo pusty, a zbiór Q obejmuje wszystkie wierzchołki grafu. W zbiorze S znajdują się wierzchołki przetworzone przez algorytm Dijkstry, a w zbiorze Q będą wierzchołki wciąż czekające na przetworzenie.</p> | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| <div></div> | <table><tr><td>u</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr><tr><td>$d[u]$</td><td>0</td><td>∞</td><td>∞</td><td>∞</td><td>∞</td><td>∞</td></tr><tr><td>$p[u]$</td><td>-</td><td>-</td><td>-</td><td>-</td><td>-</td><td>-</td></tr><tr><td></td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td></tr></table> | u | 0 | 1 | 2 | 3 | 4 | 5 | $d[u]$ | 0 | ∞ | ∞ | ∞ | ∞ | ∞ | $p[u]$ | - | - | - | - | - | - | | 1 | 1 | 1 | 1 | 1 | 1 | <p>Tworzymy dwie tablice d i p o n elementach, gdzie n oznacza liczbę wierzchołków w grafie. Elementy tablicy d będą zawierały minimalne koszty dojścia do poszczególnych wierzchołków z wierzchołka startowego. Początkowo w elementach d umieszczamy wartość +nieskończoność. W elemencie $d[v]$ umieszczamy zero.</p> <p>Elementy tablicy p będą przechowywały numery wierzchołków-poprzedników na ścieżce od wierzchołka v. Idąc wstecz po tych numerach dotrzemy do wierzchołka startowego. We wszystkich elementach tablicy p umieszczamy wartość, która nie może oznaczać numeru wierzchołka, np. -1.</p> |
| u | 0 | 1 | 2 | 3 | 4 | 5 | | | | | | | | | | | | | | | | | | | | | | | | |
| $d[u]$ | 0 | ∞ | ∞ | ∞ | ∞ | ∞ | | | | | | | | | | | | | | | | | | | | | | | | |
| $p[u]$ | - | - | - | - | - | - | | | | | | | | | | | | | | | | | | | | | | | | |
| | 1 | 1 | 1 | 1 | 1 | 1 | | | | | | | | | | | | | | | | | | | | | | | | |
| <div></div> | <table><tr><td>u</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr><tr><td>$d[u]$</td><td>0</td><td>3</td><td>∞</td><td>∞</td><td>3</td><td>∞</td></tr><tr><td>$p[u]$</td><td>-</td><td>0</td><td>-</td><td>-</td><td>0</td><td>-</td></tr><tr><td></td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td></tr></table> | u | 0 | 1 | 2 | 3 | 4 | 5 | $d[u]$ | 0 | 3 | ∞ | ∞ | 3 | ∞ | $p[u]$ | - | 0 | - | - | 0 | - | | 1 | 0 | 1 | 1 | 0 | 1 | <p>W zbiorze Q szukamy wierzchołka u o najmniejszym koszcie dojścia $d[u]$. Oczywiście, jest to wierzchołek startowy 0, dla którego $d[0] = 0$. Wierzchołek 0 przenosimy ze zbioru Q do S. Następnie przeglądamy wszystkich sąsiadów przeniesionego wierzchołka – tutaj są to wierzchołki 1 i 4. Sprawdzamy, czy:</p> |
| u | 0 | 1 | 2 | 3 | 4 | 5 | | | | | | | | | | | | | | | | | | | | | | | | |
| $d[u]$ | 0 | 3 | ∞ | ∞ | 3 | ∞ | | | | | | | | | | | | | | | | | | | | | | | | |
| $p[u]$ | - | 0 | - | - | 0 | - | | | | | | | | | | | | | | | | | | | | | | | | |
| | 1 | 0 | 1 | 1 | 0 | 1 | | | | | | | | | | | | | | | | | | | | | | | | |

| | | <p>$d[1] > d[0] + 3$? – TAK, zatem $d[1] \leftarrow d[0] + 3 = 3$. Do $p[1]$ wpisujemy 0, czyli poprzednik na ścieżce.</p> <p>$d[4] > d[0] + 3$? – TAK, zatem $d[4] \leftarrow d[0] + 3 = 3$. Do $p[4]$ wpisujemy 0, czyli poprzednik na ścieżce.</p> | | | | | | | | | | | | | | | | | | | | | |
|--------|---|---|---|----------|---|----------|---|---|--------|---|---|---|----------|---|----------|--------|----|---|---|----|---|---|--|
| | <table><tr><th>u</th><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr><tr><th>$d[u]$</th><td>0</td><td>3</td><td>4</td><td>∞</td><td>3</td><td>∞</td></tr><tr><th>$p[u]$</th><td>-</td><td>0</td><td>1</td><td>-</td><td>0</td><td>1</td></tr></table> | u | 0 | 1 | 2 | 3 | 4 | 5 | $d[u]$ | 0 | 3 | 4 | ∞ | 3 | ∞ | $p[u]$ | - | 0 | 1 | - | 0 | 1 | <p>Ponownie w zbiorze Q szukamy wierzchołka u o najmniejszym koszcie dojścia $d[u]$. Są dwa takie wierzchołki: 1 i 4 ($d[1] = 3$ i $d[4] = 3$). Wybieramy dowolny z nich, niech będzie to wierzchołek 1. Przenosimy go do zbioru S i sprawdzamy sąsiada 2:</p> <p>$d[2] > d[1] + 1$? – TAK, zatem $d[2] \leftarrow d[1] + 1 = 4$. Do $p[2]$ wpisujemy 1, czyli poprzednik na ścieżce.</p> |
| u | 0 | 1 | 2 | 3 | 4 | 5 | | | | | | | | | | | | | | | | | |
| $d[u]$ | 0 | 3 | 4 | ∞ | 3 | ∞ | | | | | | | | | | | | | | | | | |
| $p[u]$ | - | 0 | 1 | - | 0 | 1 | | | | | | | | | | | | | | | | | |
| | <table><tr><th>u</th><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr><tr><th>$d[u]$</th><td>0</td><td>3</td><td>4</td><td>∞</td><td>3</td><td>∞</td></tr><tr><th>$p[u]$</th><td>-1</td><td>0</td><td>1</td><td>-1</td><td>0</td><td>1</td></tr></table> | u | 0 | 1 | 2 | 3 | 4 | 5 | $d[u]$ | 0 | 3 | 4 | ∞ | 3 | ∞ | $p[u]$ | -1 | 0 | 1 | -1 | 0 | 1 | <p>Kolejnym wierzchołkiem u w zbiorze Q o najniższym koszcie dojścia $d[u]$ jest wierzchołek 4 ($d[4] = 3$). Przenosimy go do zbioru S, a następnie sprawdzamy koszt dojścia do jego sąsiada, wierzchołka 5:</p> <p>$d[5] > d[4] + 2$? – TAK, zatem $d[5] \leftarrow d[4] + 2 = 5$. Do $p[5]$ wpisujemy 4, czyli poprzednik na ścieżce.</p> |
| u | 0 | 1 | 2 | 3 | 4 | 5 | | | | | | | | | | | | | | | | | |
| $d[u]$ | 0 | 3 | 4 | ∞ | 3 | ∞ | | | | | | | | | | | | | | | | | |
| $p[u]$ | -1 | 0 | 1 | -1 | 0 | 1 | | | | | | | | | | | | | | | | | |
| | <table><tr><th>u</th><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr><tr><th>$d[u]$</th><td>0</td><td>3</td><td>4</td><td>7</td><td>3</td><td>5</td></tr><tr><th>$p[u]$</th><td>-1</td><td>0</td><td>1</td><td>2</td><td>0</td><td>4</td></tr></table> | u | 0 | 1 | 2 | 3 | 4 | 5 | $d[u]$ | 0 | 3 | 4 | 7 | 3 | 5 | $p[u]$ | -1 | 0 | 1 | 2 | 0 | 4 | <p>Teraz ze zbioru Q przenosimy do S wierzchołek 2 ($d[2] = 4$). Wierzchołek ten posiada w zbiorze Q dwóch sąsiadów: 3 i 5. Sprawdzamy ich koszty dojścia:</p> <p>$d[3] > d[2] + 3$? – TAK, zatem $d[3] \leftarrow d[2] + 3 = 7$. Do $p[3]$ wpisujemy 2, czyli poprzednik na ścieżce.</p> <p>$d[5] > d[2] + 1$? – NIE, nic dalej nie robimy z tym wierzchołkiem</p> |
| u | 0 | 1 | 2 | 3 | 4 | 5 | | | | | | | | | | | | | | | | | |
| $d[u]$ | 0 | 3 | 4 | 7 | 3 | 5 | | | | | | | | | | | | | | | | | |
| $p[u]$ | -1 | 0 | 1 | 2 | 0 | 4 | | | | | | | | | | | | | | | | | |
| | <table><tr><th>u</th><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr><tr><th>$d[u]$</th><td>0</td><td>3</td><td>4</td><td>6</td><td>3</td><td>5</td></tr><tr><th>$p[u]$</th><td>-1</td><td>0</td><td>1</td><td>5</td><td>0</td><td>4</td></tr></table> | u | 0 | 1 | 2 | 3 | 4 | 5 | $d[u]$ | 0 | 3 | 4 | 6 | 3 | 5 | $p[u]$ | -1 | 0 | 1 | 5 | 0 | 4 | <p>Do zbioru S przenosimy wierzchołek 5 ($d[5] = 5$). W zbiorze Q ma on tylko jednego sąsiada, wierzchołek 3. Sprawdzamy koszt dojścia do wierzchołka 3:</p> <p>$d[3] > d[5] + 1$? – TAK, zatem $d[3] \leftarrow d[5] + 1 = 6$. Do $p[3]$ wpisujemy 5, czyli poprzednik na ścieżce.</p> <p>Zwróć uwagę, że w tym przypadku algorytm znalazł lepszą ścieżkę do wierzchołka 3 o niższym koszcie.</p> |
| u | 0 | 1 | 2 | 3 | 4 | 5 | | | | | | | | | | | | | | | | | |
| $d[u]$ | 0 | 3 | 4 | 6 | 3 | 5 | | | | | | | | | | | | | | | | | |
| $p[u]$ | -1 | 0 | 1 | 5 | 0 | 4 | | | | | | | | | | | | | | | | | |

| u | 0 | 1 | 2 | 3 | 4 | 5 |
|--------|----|---|---|---|---|---|
| $d[u]$ | 0 | 3 | 4 | 6 | 3 | 5 |
| $p[u]$ | -1 | 0 | 1 | 5 | 0 | 4 |

Do zbioru S przenosimy ostatni wierzchołek 3. Zbiór Q stał się pusty, zatem przeniesiony wierzchołek nie ma w zbiorze Q żadnych sąsiadów. Algorytm kończy się. W tablicy d mamy policzone koszty dojścia do każdego z wierzchołków grafu. W tablicy p znajdują się poprzedniki na ścieżce każdego wierzchołka, co pozwala odtworzyć ścieżki do poszczególnych wierzchołków grafu: idziemy wstecz po kolejnych poprzednikach, aż dojdziemy do wierzchołka startowego, dla którego nie istnieje poprzednik (wartość -1).

Z tablic d i p odczytujemy:

- Dojście do wierzchołka 0: ścieżka pusta, koszt 0
- Dojście do wierzchołka 1: 0–1, koszt 3
- Dojście do wierzchołka 2: 0–1–2, koszt 4
- Dojście do wierzchołka 3: 0–4–5–3, koszt 6
- Dojście do wierzchołka 4: 0–4, koszt 3
- Dojście do wierzchołka 5: 0–4–5, koszt 5

11. Zadania