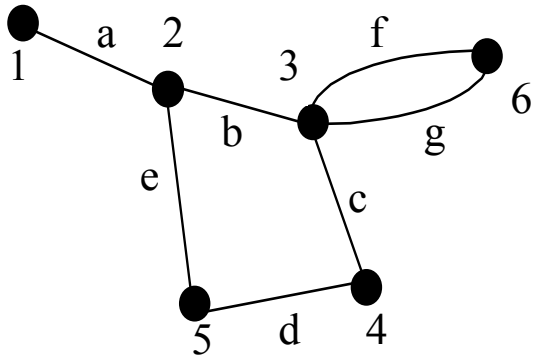
Wykład 10
Grafy, algorytmy grafowe

1. Typy złożoności obliczeniowej

Typ złożoności	oznaczenie	n		
Jedna operacja trwa 1μs		10	50	100
logarytmiczna	lgn	0.000003 s	0.000006 s	0.000007 s
liniowa	n	0.00001 s	0.00005 s	0.0001 s
Logarytmiczno-liniowa	nlgn	0.000033 s	0.000282 s	0.000664 s
kwadratowa	n^2	0.0001 s	0.0025 s	0.01 s
Wielowamianowa (k-stopien wielomianu)	n^k k=3	0.001 s	0.008 s	1 s
wykładnicza	2^n	0.001024 s	35.6 lat	$4.0087*10^{16}$ lat
wykładnicza n!	n!	3.6288 s	$2.63136*10^{68}$ lat	$2.9613*10^{144}$ lat

2. Reprezentacje grafu

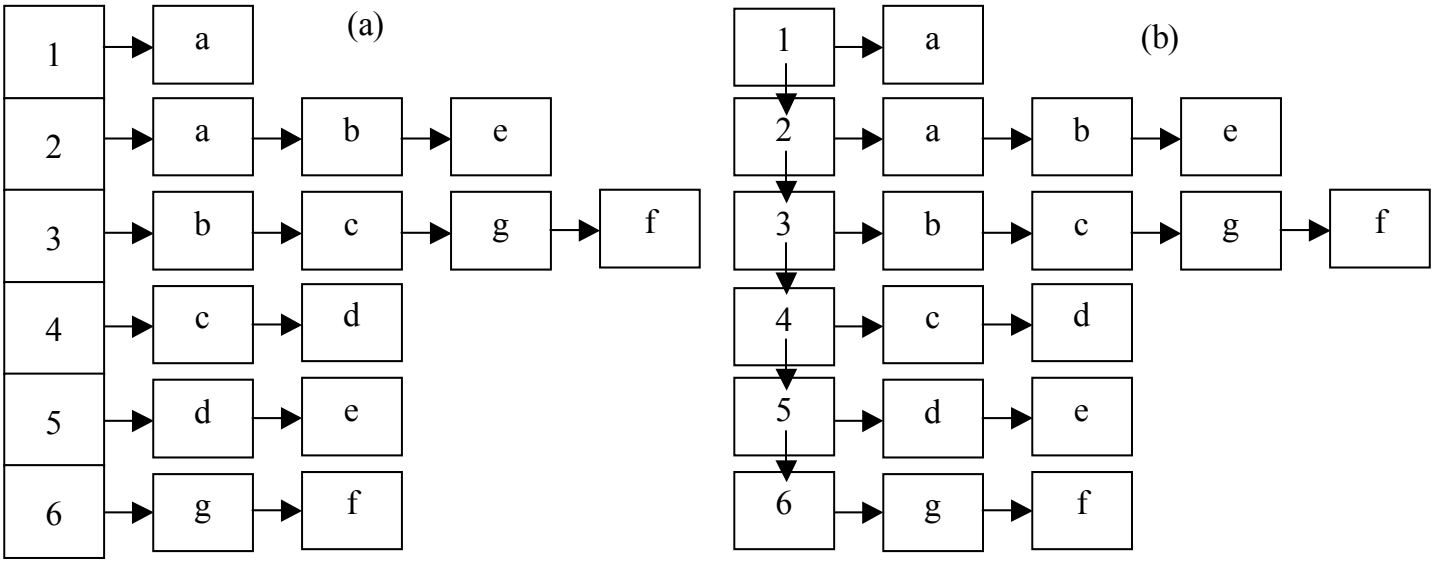
Macierz incydencji – w programach przedstawiana jako tablica dwuwymiarowa, gdzie indeksy wierszy reprezentują numery wierzchołków, a indeksy kolumn oznaczają numery krawędzi. Elementy równe 1 oznaczają krawędzie oznaczone numerami kolumn incydentne z wierzchołkiem oznaczonym numerem wiersza



	a	*b*	*c*	*d*	*e*	*f*	*g*
1	1	0	0	0	0	0	0
2	1	1	0	0	1	0	0
3	0	1	1	0	0	1	1
4	0	0	1	1	0	0	0
5	0	0	0	1	1	0	0
6	0	0	0	0	0	1	1

Lista incydencji jest efektywną postacią macierzy incydencji – w programach reprezentowana przez:

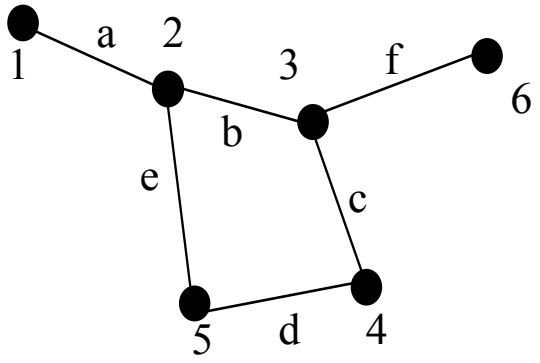
- tablice struktur, które zawierają numer wierzchołka oraz wskaźnik na listę krawędzi incydentnych do danych wierzchołków; lista krawędzi incydentnych zawiera elementy, które mają numer wierzchołka i wskaźnik na taki element (a)
- lista elementów zawierających numer wierzchołka, wskaźnik na listę krawędzi incydentnych do danych wierzchołków (jw) oraz wskaźnik na taki element (b)



Własności macierzy incydencji:

1. Jeśli każda krawędź jest incydentna do dwóch wierzchołków, to w każdej kolumnie mamy tylko dwie jedynki
2. Wiersz z samych zer reprezentuje wierzchołek izolowany
3. Pętla jest reprezentowana przez jedną jedynkę
4. Krawędzie równoległe tworzą dwie identyczne kolumny w macierzy
5. Jeśli graf jest niespójny i składa się np. z 2 składowych g1 i g2, to można je wyrazić za pomocą części macierzy, które nie mają wspólnych wierszy i kolumn
6. Dwa grafy są izomorficzne, jeśli różnią się tylko permutacją wierszy i kolumn, czyli jedynie różnymi etykietami wierzchołków i krawędzi

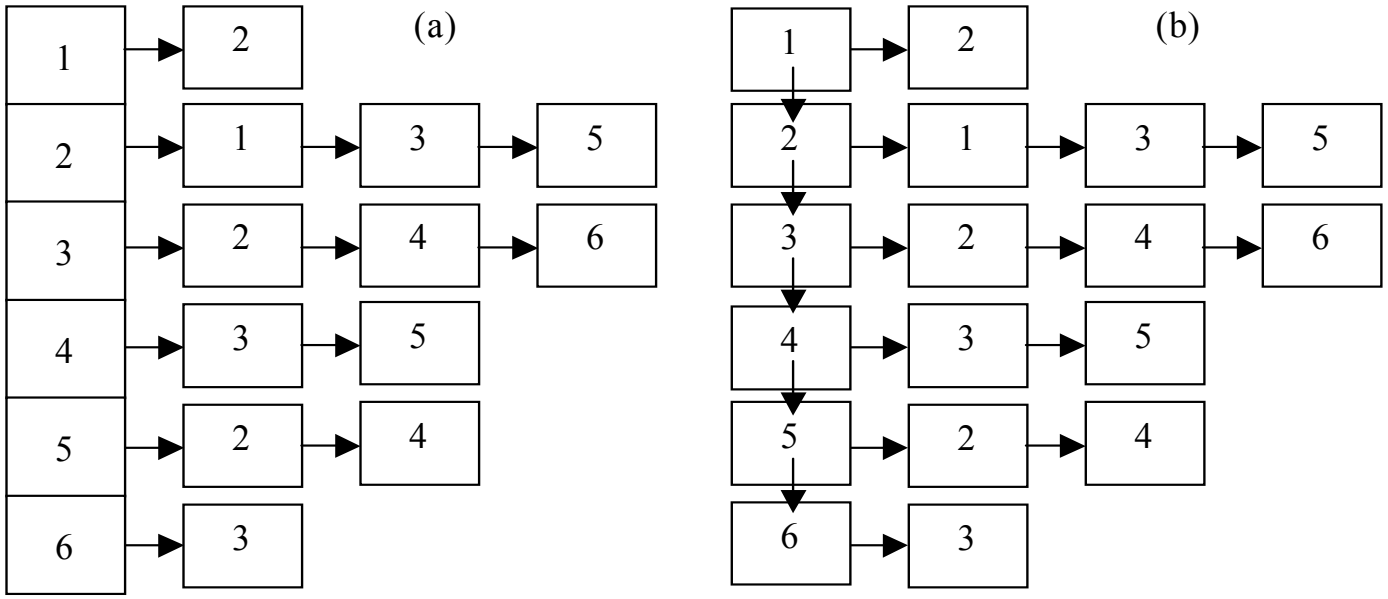
Macierz sąsiedztwa grafu nieskierowanego - w programach przedstawiana jako tablica dwuwymiarowa, gdzie indeksy wierszy i kolumn reprezentują numery wierzchołków, a wartości elementów równe 1 oznaczają krawędź łącząca wierzchołki określone numerem wiersza i kolumny



	1	*2*	*3*	*4*	*5*	*6*
1	0	1	0	0	0	0
2	1	0	1	0	1	0
3	0	1	0	1	0	1
4	0	0	1	0	1	0
5	0	1	0	1	0	0
6	0	0	1	0	0	0

Lista sąsiedztwa jest efektywną pamięciowo postacią macierzy sąsiedztwa – w programach reprezentowana przez:

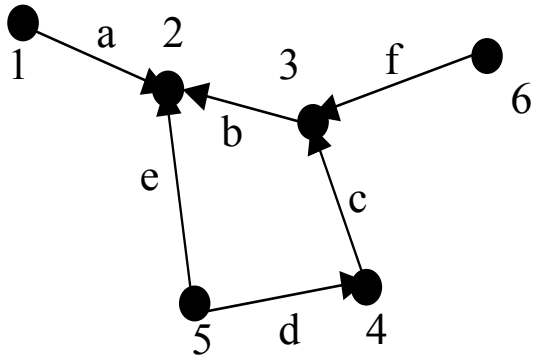
- tablice struktur, które zawierają numer wierzchołka oraz wskaźnik na listę wierzchołków sąsiadującymi z danym wierzchołkiem; lista wierzchołków sąsiadujących zawiera elementy, które mają numer wierzchołka oraz wskaźnik na taki element (a)
- lista elementów zawierających numer wierzchołka, wskaźnik na listę wierzchołków sąsiadującymi z danym wierzchołkiem (jw) oraz wskaźnik na taki element (b)



Własność **macierzy sąsiedztwa**

1. Elementy wzdłuż głównej przekątnej są równe 0, jeśli graf nie ma pętli, oraz 1 jeśli ma pętle
2. Nie można wyrazić krawędzi równoległych
3. Wiersz i kolumny muszą być ustawione w tym samym porządku. Oznacza to, że przestawienie dwóch wierszy wymaga przestawienia odpowiadających kolumn
4. Jeśli graf jest niespójny i składa się np. z 2 składowych g1 i g2, to można je wyrazić za pomocą części macierzy, które nie mają wspólnych wierszy i kolumn

Macierz sąsiedztwa grafu skierowanego - w programach przedstawiana jako tablica dwuwymiarowa, gdzie indeksy wierszy i kolumn reprezentują numery wierzchołków, a wartości elementów równe 1 oznaczają krawędź łącząca wierzchołki określone numerem wiersza i kolumny



	1	*2*	*3*	*4*	*5*	*6*
1	0	1	0	0	0	0
2	0	0	0	0	0	0
3	0	1	0	0	0	0
4	0	0	1	0	0	0
5	0	1	0	1	0	0
6	0	0	1	0	0	0

Lista sąsiedztwa jest efektywną pamięciowo postacią macierzy sąsiedztwa – w programach reprezentowana przez:

- tablice struktur, które zawierają numer wierzchołka oraz wskaźnik na listę wierzchołków sąsiadującymi z danym wierzchołkiem (zgodnie z kierunkiem krawędzi); lista wierzchołków sąsiadujących zawiera elementy, które mają numer wierzchołka oraz wskaźnik na taki element (a)
- lista elementów zawierających numer wierzchołka, wskaźnik na listę wierzchołków sąsiadującymi z danym wierzchołkiem (zgodnie z kierunkiem krawędzi) oraz wskaźnik na taki element (b)

3. Algorytmy grafowe

3.1. Przeszukiwanie grafu „w głąb” DFS - algorytmy „z powrotami”

Algorytmy z powrotami są wykorzystywane do rozwiązywania problemów, w których z określonego **zbioru** jest wybierana **sekwencja** obiektów tak, aby spełniała ona określone **kryteria**.

Przykład 1

Przeszukiwanie drzewa binarnego w sposób przedrostkowy jest przykładem **przeszukiwania w głąb** czyli przykładem algorytmu z powrotami (wykład 8)

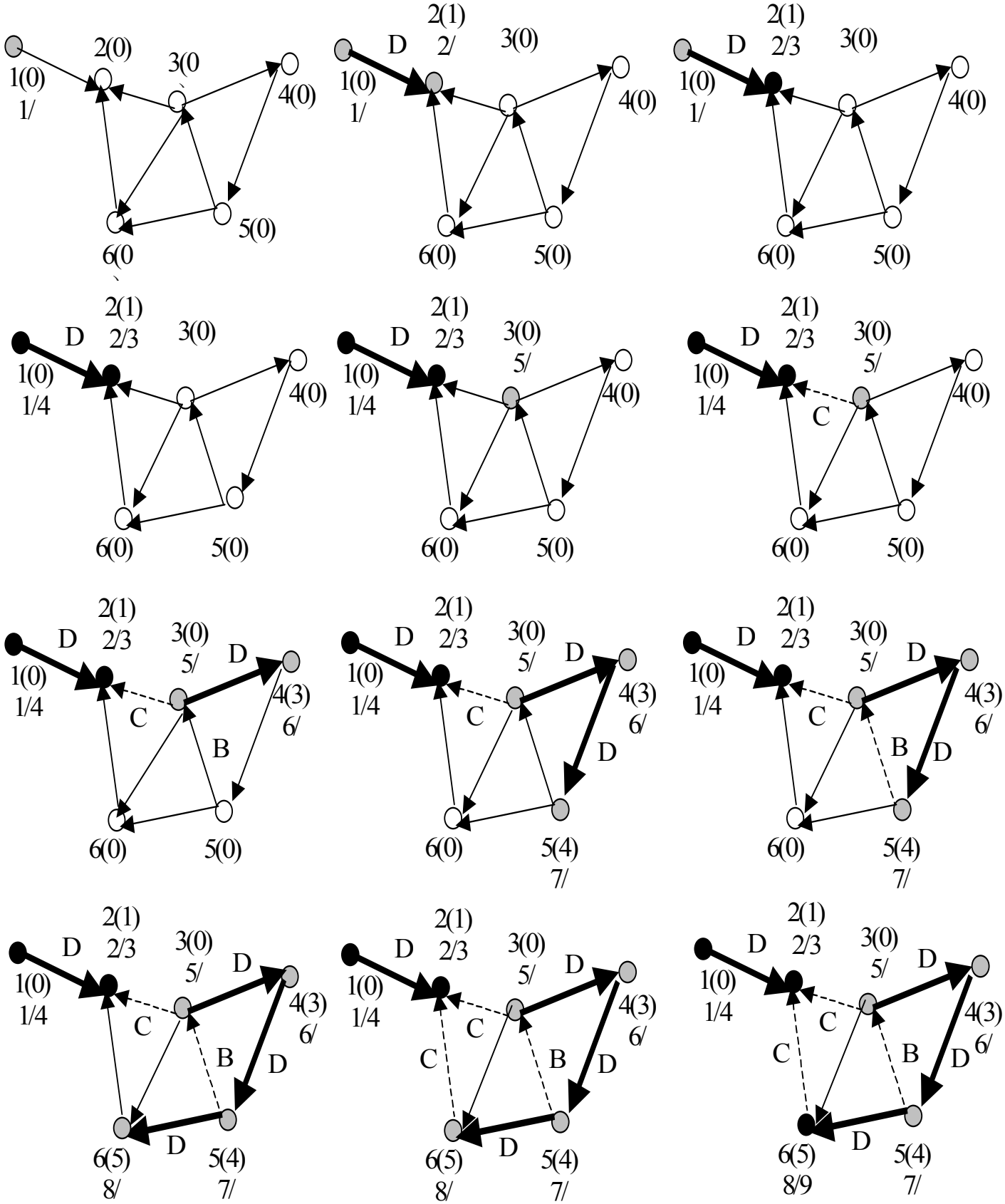
Przykład 2

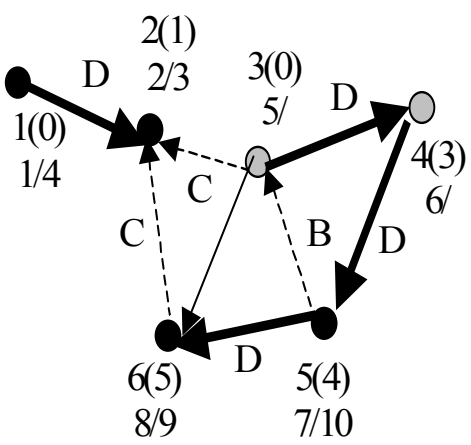
Wynikiem przeszukiwanie grafu w głąb jest las złożony z jednego lub wielu drzew przeszukiwania w głąb (gdy graf jest niespójny lub jest grafem skierowanym), który zawiera wszystkie wierzchołki grafu.

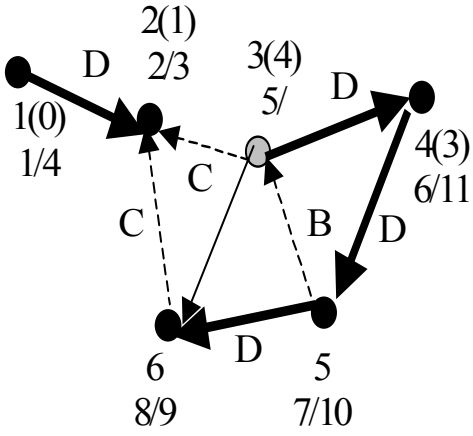
G= {V, E}, gdzie V jest zbiorem wierzchołków (węzłów) i E zbiorem krawędzi
BIALY – kolor nadawany nie odwiedzonym wierzchołkom
SZARY – kolor nadawany odwiedzonym wierzchołkom
CZARNY – kolor nadawany węzłowi, którego lista sąsiedztwa została odwiedzona

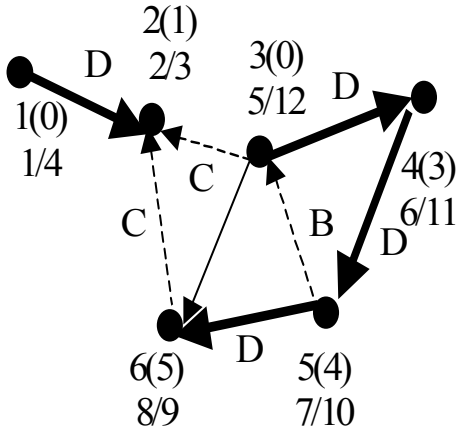
```
wierzcholek
  {  kolor;
     odwiedzony;
     czas_1;
     czas_2;}

wierzcholek wierzcholki[V[G]]

DFS (G)
   1. dla każdego wierzchołka  u należącego do V[G]
        { wierzcholki[u].kolor = BIALY
          wierzcholki[u].odwiedzony = 0 }
   2. czas = 0
   3. dla każdego wierzcholka u należącego do V[G]
            jeśli  wierzcholki[u].kolor == BIALY  //nowy wierzchołek drzewa w lesie przeszukiwań
            to wywołaj DFS_VISIT(u)               // lub nowe drzewo poszukiwań

DFS_VISIT(u)
   1. wierzcholki[u].kolor = SZARY
   2. wierzcholki[u].czas_1 = ++czas
   3. dla każdego wierzchołka v należącego do listy sąsiedztwa wierzchołka u
          {  jeśli  wierzcholki[v].kolor == BIALY
             to     wierzcholki[v].odwiedzony = u   //u -poprzednik nie odwiedzonego następcy v
             DFS_VISIT(v) }
   4. wierzcholki[u].kolor = CZARNY
   5. wierzcholki[u].czas_2 = ++czas
```

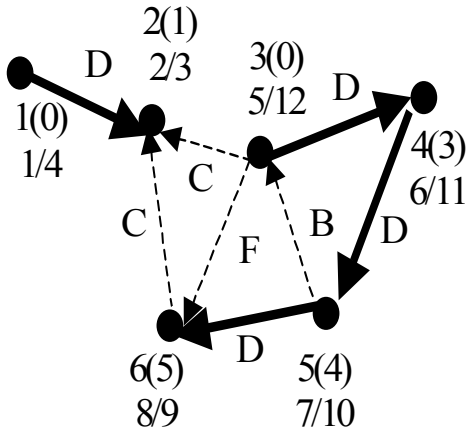
Graf skierowany: D: krawędź drzewa przeszukiwania (krawędź (u, v), gdy odwiedzony został v), B: krawędź powrotna (również dla krawędzi typu pętla – łączy v z jego przodkiem u), F: krawędź w przód (łączy u z potomkiem v), C: krawędź poprzeczna (prowadząca do innego drzewa poszukiwań w grafie skierowanym lub niespójnym lub łączy wierzchołki, z których jeśli tylko jeden nie jest przodkiem drugiego),



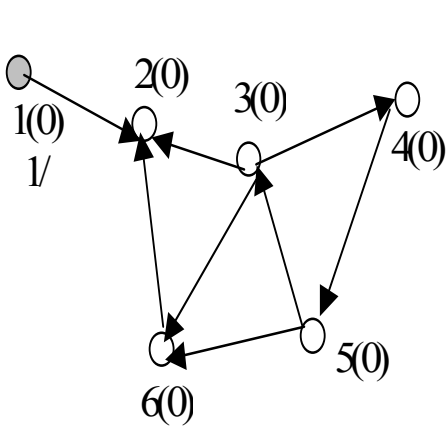


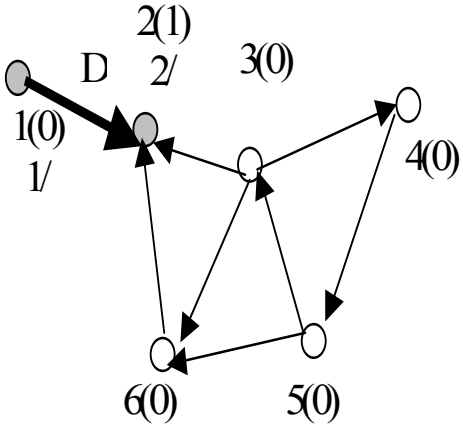


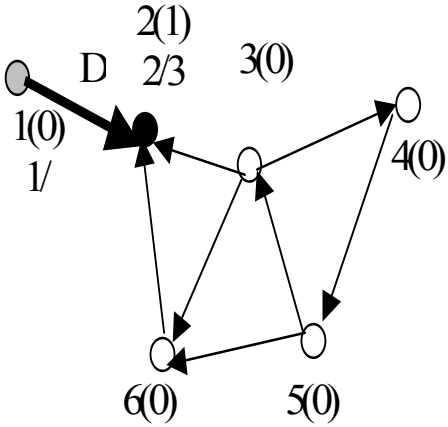


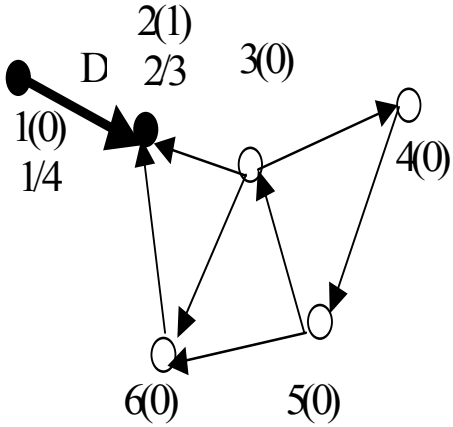


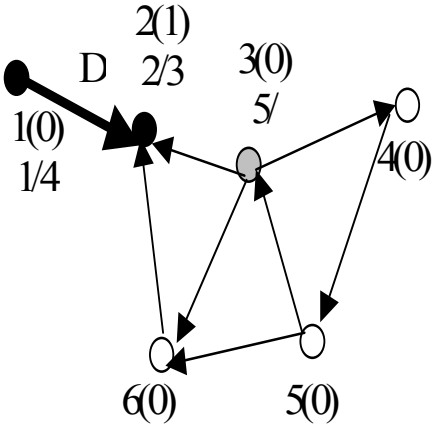
Drugi wariant

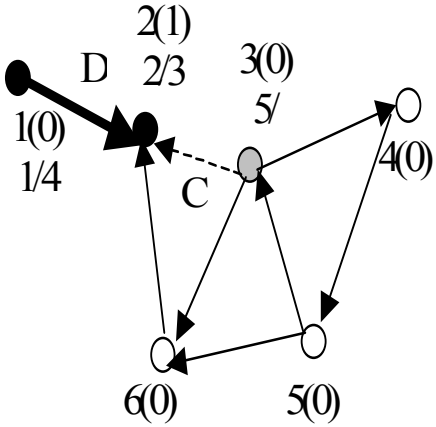


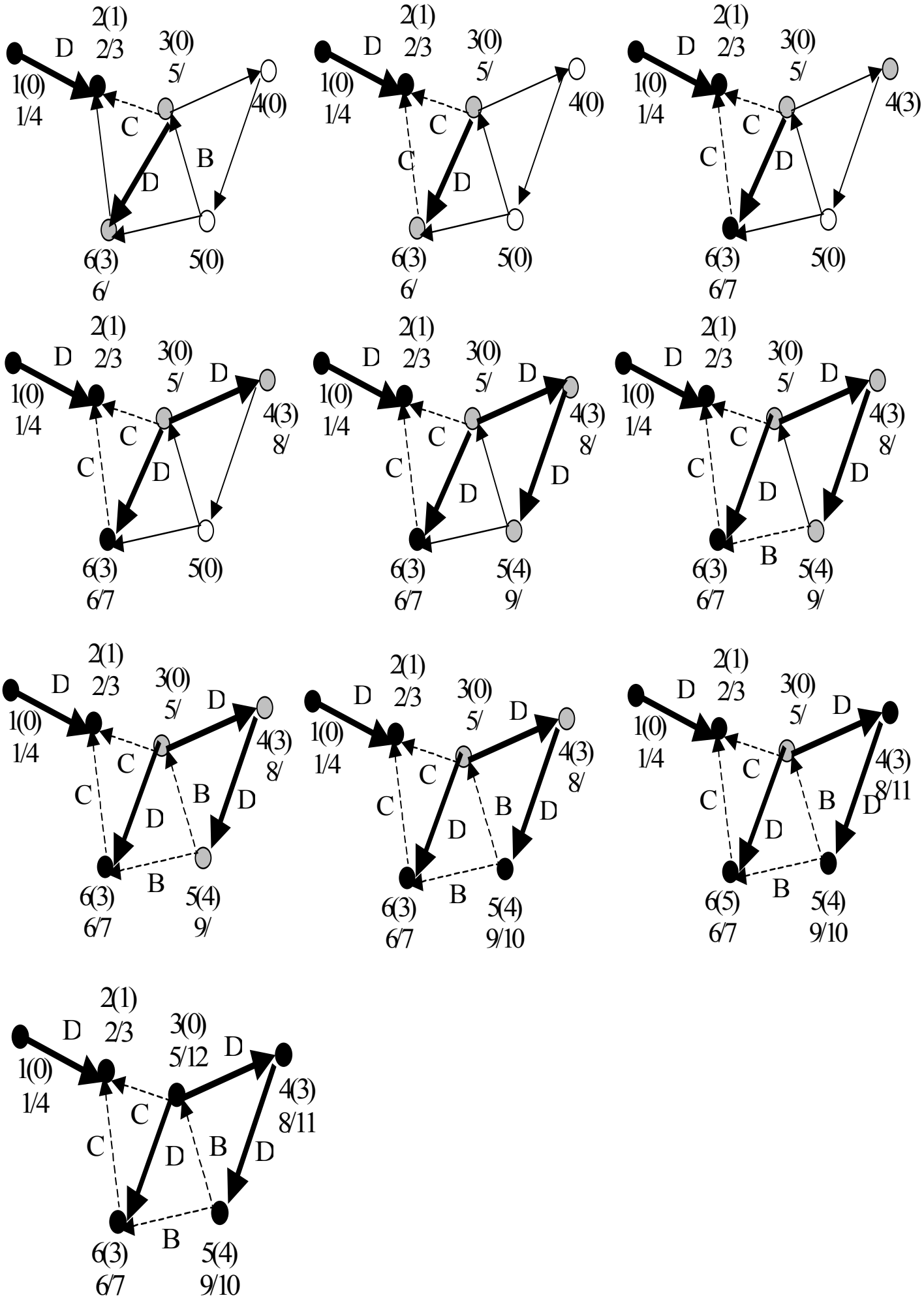











2(1)
2/3
3(0)
5/
1(0)
1/4
4(0)
6(3)
6/
5(0)
4(3)
8/
5(4)
9/
6(5)
6/7
9/10
3(0)
5/12
4(3)
8/11
D
C
B

Wypisywanie wierzchołków ścieżki z s do v przy założeniu, że drzewa przeszukiwań zostały wcześniej zbudowane za pomocą procedury DFS:

```
Drukuj (G, s , v)
1. jeśli (v == s)  to wypisz (s)
   lub
     jeśli (wierzcholki[v].odwiedzony == 0)
        to komunikat (”Brak scieżki z s do v”)
     lub
        { Drukuj(G, s, wierzcholki[v].odwiedzony)
          wypisz(v) }
```

Złożoność obliczeniowa procedury DFS

Czas inicjowania tablicy (linia 1 procedury) oraz czas wybierania kolejnego wierzchołka (linia 3 procedury) jest równy O(V).
Łączny czas odwiedzania listy sąsiedztwa każdego wierzchołka białego jest równy O(E) - czas procedury DFS_VISIT, gdzie E jest zbiorem krawędzi grafu i odwiedzane są jedynie białe wierzchołki w liście sąsiedztwa. Każde przeszukanie pojedynczej listy sąsiedztwa jest wykonywane tylko raz dla każdego wierzchołka należącego do zbioru wierzchołków V. Stąd czas działania procedury DFS jest równy O(V+E).

Uwaga:
Inne przykłady rozwiązywane metodą **algorytmów z powrotami**

- Problem n królowych (ustawienie takie, aby królowe się nie szachowały),
- problem sumy podzbiorów (znalezienie wśród zbioru przedmiotów, podzbioru o zadanej wadze W),
- kolorowanie grafu nieskierowanego (znalezienie wszystkich sposobów pokolorowania wierzchołków przy zastosowaniu co najwyżej m kolorów, aby sąsiednie wierzchołki nie miały tego samego koloru),
- problem cyklu Hamiltona (wyszukanie w grafie nieskierowanym ścieżki, która rozpoczyna się na wybranym wierzchołku, odwiedza każdy wierzchołek grafu dokładnie jeden raz i kończy się na pierwszym wierzchołku),
- optymalizacyjny problem plecakowy (przy założonej wadze plecaka maksymalizacja wartości chowanych przedmiotów),

3.2. Algorytm przeszukiwania wszerz BFS – obliczanie najkrótszej ścieżki z s do wszystkich osiągalnych wierzchołków metodą podziału i ograniczeń

Jest to usprawnienie algorytmu z powrotami. Nie narzuca on żadnego określonego sposobu przeglądania drzewa oraz jest wykorzystywany jedynie do problemów optymalizacyjnych.
Oblicza on w każdym węźle liczbę (granicę), która pozwoli stwierdzić, czy ten węzeł jest **obiecujący**. Liczba ta jest granicą wartości rozwiązania, jakie może zostać uzyskane dzięki rozwinięciu węzła. Jeśli ta granica nie jest lepsza niż wartość dotychczas znalezionego rozwiązania, węzeł jest **nieobiecujący**. Wartość optymalna jest albo minimalna albo maksymalna, czyli wartość lepsza oznacza tu albo większą albo mniejszą wartość. W najgorszym wypadku algorytmy te mają złożoność wykładniczą lub gorszą od wykładniczej.

G= {V, E}, gdzie V jest zbiorem wierzchołków (węzłów) i E zbiorem krawędzi
BIALY – kolor nadawany nie odwiedzonym wierzchołkom
SZARY – kolor nadawany odwiedzonym wierzchołkom
CZARNY – kolor nadawany węzłowi, którego lista sąsiedztwa została odwiedzona

```
wierzcholek
  {  kolor;
     odwiedzony;
     czas;}

wierzcholek wierzcholki[V[G]]
```

BFS (G)

1. dla każdego wierzchołka u należącego do V[G]
   { wierzcholki[u].kolor = BIALY
     wierzcholki[u].czas = -1
     wierzcholki[u].odwiedzony = 0 }
2. wierzcholki[s].kolor = SZARY
3. wierzcholki[s].czas = 0
4. Inicjuj (Kolejka)
5. Wstaw(Kolejka, s)
6. Dopóki !Pusta(Kolejka)
   6.1. u = Usun(Kolejka) //u nie jest usuwany z kolejki, lecz odczytany z jej początku
   6.2. dla każdego wierzcholka v należącego do listy sąsiedztwa wierzchołka u
        jeśli wierzcholki[v].kolor == BIALY
        to
        { wierzcholki[v].kolor = SZARY
          wierzcholki[v].czas = wierzcholki[u].czas +1
          wierzcholki[v].odwiedzony = u
          Wstaw(Kolejka, v) }
   6.3. Usun(Kolejka) //usuwany pierwszy element z kolejki
   6.4. wierzcholki[u].kolor = CZARNY

Wypisywanie wierzchołków najkrótszej ścieżki z s do v przy założeniu, że drzewo najkrótszych ścieżek zostało wcześniej zbudowane za pomocą procedury BFS:

```
Drukuj (G, s , v)
2. jeśli (v == s)  to wypisz (s)
   lub
     jeśli (wierzcholki[v].odwiedzony == 0)
        to komunikat (”Brak scieżki z s do v”)
     lub
        { Drukuj(G, s, wierzcholki[v].odwiedzony)
          wypisz(v) }
```

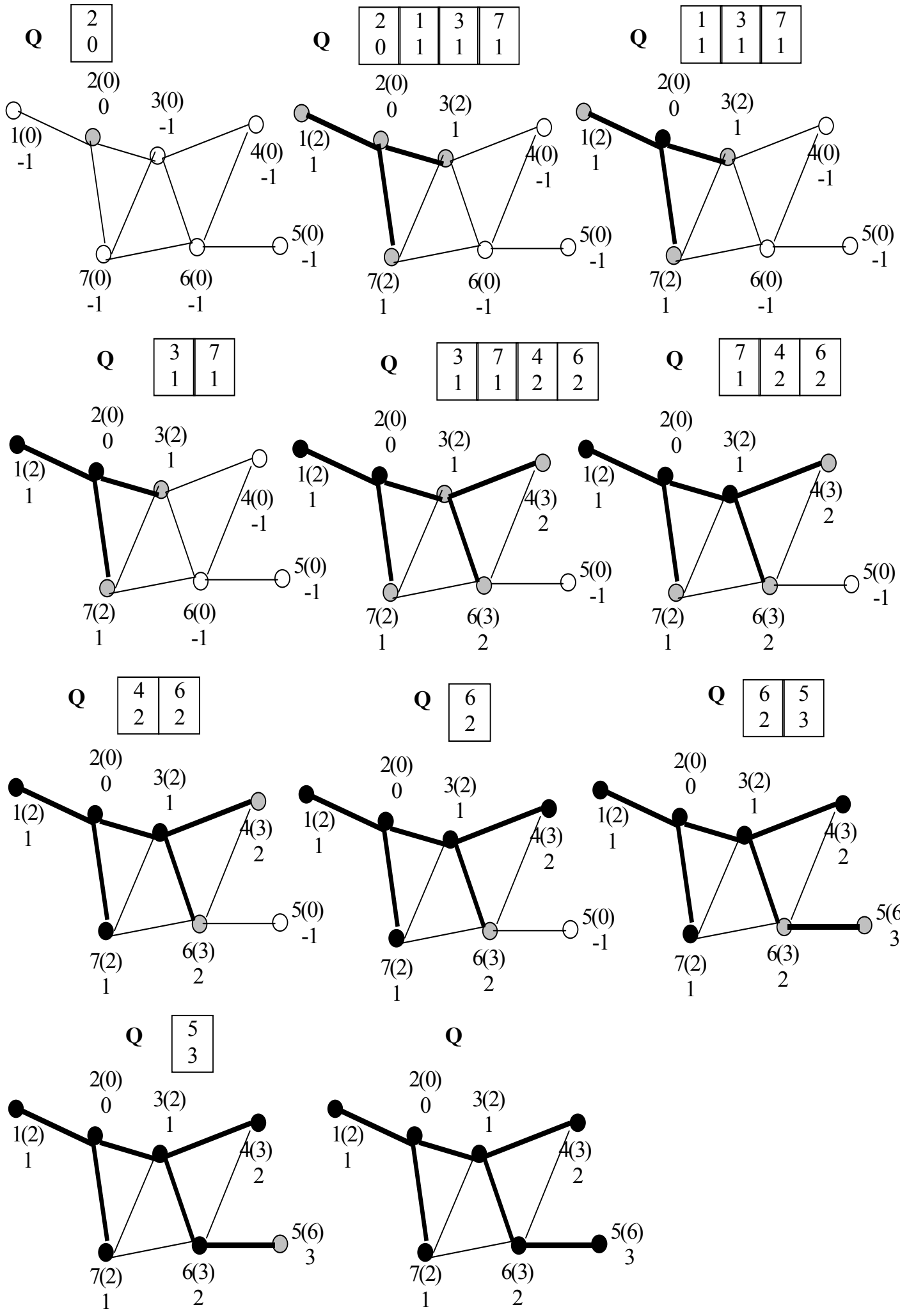
Złożoność obliczeniowa:

1) Czas operacji na kolejce jest równy O(V). Po każdym przeglądaniu listy sąsiedztwa pojedynczego wierzchołka jest on usuwany z kolejki. Łączny czas przeglądania listy sąsiedztwa każdego wierzchołka jest równe O(E). Inicjowanie trwa O(V). Czas procedury BFS wynosi O (V + E).

2) Procedura Drukuj (G, s, v) działa w czasie liniowym, gdyż każde wywołanie rekurencyjne następuje dla ścieżki krótszej o jeden wierzchołek.

Uwaga:
Inne przykłady rozwiązywane **metodą podziału i ograniczeń**

- problem plecakowy (przy założonej wadze plecaka maksymalizacja wartości chowanych przedmiotów)
- problem komiwojażera (znalezienie najkrótszej ścieżki w grafie skierowanym, rozpoczynającej się w danym wierzchołku, odwiedzającej wszystkie wierzchołki dokładnie raz i kończącej się w wierzchołku początkowym)


Q
2
0
2(0)
0
3(0)
-1
1(0)
-1
4(0)
-1
5(0)
-1
7(0)
-1
6(0)
-1
Q
2
0
1
1
3
1
7
1
2(0)
0
3(2)
1
1(2)
1
4(0)
-1
5(0)
-1
7(2)
1
6(0)
-1
Q
1
1
3
1
7
1
2(0)
0
3(2)
1
1(2)
1
4(0)
-1
5(0)
-1
7(2)
1
6(0)
-1
Q
3
1
7
1
2(0)
0
3(2)
1
1(2)
1
4(0)
-1
5(0)
-1
7(2)
1
6(0)
-1
Q
3
1
7
1
4
2
6
2
2(0)
0
3(2)
1
1(2)
1
4(3)
2
5(0)
-1
7(2)
1
6(3)
2
Q
7
1
4
2
6
2
2(0)
0
3(2)
1
1(2)
1
4(3)
2
5(0)
-1
7(2)
1
6(3)
2
Q
4
2
6
2
2(0)
0
3(2)
1
1(2)
1
4(3)
2
5(0)
-1
7(2)
1
6(3)
2
Q
6
2
2(0)
0
3(2)
1
1(2)
1
4(3)
2
5(0)
-1
7(2)
1
6(3)
2
Q
6
2
5
3
2(0)
0
3(2)
1
1(2)
1
4(3)
2
5(6)
3
7(2)
1
6(3)
2
Q
5
3
2(0)
0
3(2)
1
1(2)
1
4(3)
2
5(6)
3
7(2)
1
6(3)
2
Q
2(0)
0
3(2)
1
1(2)
1
4(3)
2
5(6)
3
7(2)
1
6(3)
2

3.3. Algorytm Floyda określania najkrótszej ścieżki w grafie – metodą programowania dynamicznego.

Programowanie dynamiczne usuwa niedogodności algorytmów typu „dziel i zwyciężaj", które wykonując podziały, mogą powtórzyć rozwiązanie tych samych realizacji i stają się wtedy bardzo niewydajne.

Przykład
1) Obliczanie liczb Fibonacciego rzędu 1 w sortowaniu polifazowym z trzema plikami metodą programowania dynamicznego

Serie w plikach			Idealne serie		Numer poziomu scalania - *l*	Liczba serii
f1	*f2*	*f3*	f_1	f_2		
13	8	0	13	8	6	21
5	0	8	8	5	5	13
0	5	3	5	3	4	8
3	2	0	3	2	3	5
1	0	2	2	1	2	3
0	1	1	1	1	1	2
1	0	0	1	0	0	1

```
int Fibonacci (int n)
{ int i, f[n+1];
 f[0]=0;
 if (n > 0)
   f[1] = 1;
 for (i = 2; i<=n; i++)
   { f[i] = f[i -1] + f[i – 2];
    }
 return f[n];
}
```

$f_1^0 = 1$, $f_2^0 = 0$, $f_2^{l+1} = f_1^l$, $f_1^{l+1} = f_1^l + f_2^l$ dla $l > 0$

Jeśli przyjmie się, że $f_1^l = f_l$, to wzory rekurencyjne definiujące ciąg Fibonacciego rzędu 1:

$f_{i+1} = f_i + f_{i-1}$ dla $i \geq 1$ oraz $f_1 = 1$, $f_0 = 0$

Wniosek: Początkowe liczby serii na dwóch plikach muszą być dwoma kolejnymi elementami ciągu Fibonacciego rzędu 1, natomiast trzeci plik służy do łączenia serii na kolejnym poziomie.

2) Obliczanie liczb Fibonacciego metodą „dziel i zwyciężaj"

```
int Fibonacci(int n)
{ if (n <= 1)
    return n;
  else
  return Fibonacci (n-1) + Fibonacci (n-2)
}
```

Aby obliczyć liczby Fibonacciego n i n+1 należy dwukrotnie obliczyć liczbę n-1 itd. Prowadzi to do złożoności wykładniczej $2^{n/2}$

n	Prog. Dyn.	Algorytm „Dziel i zwyciężaj" - dolne ograniczenie czasu
80	81 ns	$1.1 * 10^{12}$ ns = 18 min
200	201 ns	$1.3 * 10^{30}$ ns = $4 * 10^{13}$ lat

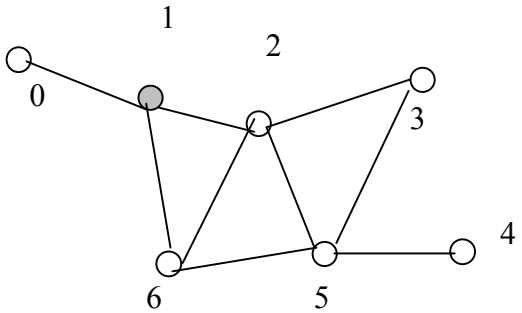
Programowanie dynamiczne polega na wykonaniu następujących działań:

1) Określa się właściwość rekurencyjną, która pozwala znaleźć rozwiązanie realizacji problemu
2) Rozwiązuje się realizację problemu zgodnie z podejściem wstępującym, najpierw rozwiązując mniejsze realizacje

Algorytm Floyda wyznaczania najkrótszej ścieżki w grafie – $\Theta(N^3)$

Zadanie: Należy obliczyć najkrótsze drogi z każdego wierzchołka w grafie ważonym do wszystkich innych wierzchołków. Wagi krawędzi są liczbami nieujemnymi. Dana jest macierz sąsiedztwa Graf o rozmiarze N x N, której elementy Graf[i][j] są wagą krawędzi prowadzącej od i-tego wierzchołka do j-tego wierzchołka lub równe 0 dla i=j oraz równe N (N oznacza nieskończoność), gdy brak krawędzi. W wyniku algorytmu otrzymuje się tablicę dwuwymiarową D o rozmiarze N x N, gdzie D[i][j] jest długością najkrótszej drogi prowadzącej z i-tego wierzchołka do j-tego wierzchołka oraz tablica dwuwymiarowa P o rozmiarze N x N, gdzie P[i][j] jest numerem pośredniego wierzchołka o najwyższym indeksie w najkrótszej drodze od i-tego wierzchołka do j-tego wierzchołka lub -1, jeśli taki wierzchołek nie istnieje.

```
void Floyd (int N, const int Graf [ ][N], int D[ ][N], int P[ ][N])
{  int i, j, k
   for (i = 0; i<N; i++)
      for (j =0; j<N; j++)
        P[i][j] = -1;
   memmove(D, Graf,N*N*sizeof(int));
   for (k = 0; k<N; k++)
        for (i = 0; i< N; i++)
          for (j=0; j<N; j++)
              if (D[i][k] +D[k][j] < D[i][j])
                  {  P[i][j] = k;
                     D[i][j]= D[i][k] + D[k][j];
                   }
 }
```



Wagi krawędzi są równe 1, N=10

Graf	0	1	2	3	4	5	6
0	**0**	**1**	**10**	**10**	**10**	**10**	**10**
1	**1**	**0**	**1**	**10**	**10**	**10**	**1**
2	**10**	**1**	**0**	**1**	**10**	**1**	**1**
3	**10**	**10**	**1**	**0**	**10**	**1**	**10**
4	**10**	**10**	**10**	**10**	**0**	**1**	**10**
5	**10**	**10**	**1**	**1**	**1**	**0**	**1**
6	**10**	**1**	**1**	**10**	**10**	**1**	**0**

D	0	1	2	3	4	5	6
0	**0**	**1**	**2**	**3**	**4**	**3**	**2**
1	**1**	**0**	**1**	**2**	**3**	**2**	**1**
2	**2**	**1**	**0**	**1**	**2**	**1**	**1**
3	**3**	**2**	**1**	**0**	**2**	**1**	**2**
4	**4**	**3**	**2**	**2**	**0**	**1**	**2**
5	**3**	**2**	**1**	**1**	**1**	**0**	**1**
6	**2**	**1**	**1**	**2**	**2**	**1**	**0**

P	0	1	2	3	4	5	6
0	**-1**	**-1**	**1**	**2**	**5**	**2**	**1**
1	**-1**	**-1**	**-1**	**2**	**5**	**2**	**-1**
2	**1**	**-1**	**-1**	**-1**	**5**	**-1**	**-1**
3	**2**	**2**	**-1**	**-1**	**5**	**-1**	**2**
4	**5**	**5**	**5**	**5**	**-1**	**-1**	**5**
5	**2**	**2**	**-1**	**-1**	**-1**	**-1**	**-1**
6	**1**	**-1**	**-1**	**2**	**5**	**-1**	**-1**

```
void drukuj(int P[ ][N], int s, int v)
    { if (P [s][v] !=-1)
       {  drukuj(P, s, P[s][v]);
          cout<<"v: "<< P[s][v];
          drukuj(P, P[s][v], v); }
    }
```