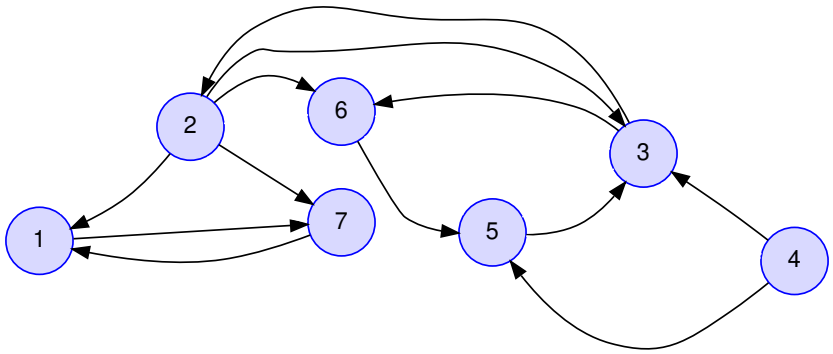
Omówienie – zestaw 4.[1]

Ad. 1. Skierowany graf losowy

Początek projektu polega na wygenerowaniu losowego digrafu według modelu $G(n, p)$. Żeby to zrealizować, program musi mieć zaimplementowane **kodowanie grafów skierowanych**, zwanych też **digrafami** (z angielskiego: *directed graph*, w skrócie *digraph*). Krawędzie digrafów – nazywane często **łukami** – są skierowane, tzn. interesuje nas ich zwrot.

W kategoriach grafów skierowanych mówi się o **digrafach prostych**: są to grafy, w których wszystkie **łuki** są różne i żaden z nich nie jest pętlą.

Grafy skierowane kodujemy według poznanych już wcześniej reprezentacji: **listy sąsiedztwa**, **macierzy sąsiedztwa** i/lub **macierzy incydencji**. Reprezentacje te zostaną omówione na przykładzie digrafu prostego zaprezentowanego na rys. 1.



Rysunek 1: Przykładowy losowy graf skierowany G. Dwie krawędzie incydentne z tymi samymi dwoma wierzchołkami u i v, ale o przeciwnych zwrotach, nie oznaczają krawędzi wielokrotnej (np. krawędzie $(1, 7)$ oraz $(7, 1)$ są różne) .

(a) **Lista sąsiedztwa** powstaje bardzo podobnie, jak w przypadku grafów nieskierowanych: dla każdego wierzchołka u digrafu wypisujemy te wierzchołki, **do których skierowane są łuki** wychodzące z u. Listing 1. przedstawia listę sąsiedztwa grafu z rys. 1.

```
1.  7
2.  1 3 6 7
3.  2 6
4.  3 5
5.  3
6.  5
7.  1
```

Listing 1: **Lista sąsiedztwa** grafu skierowanego z rys. 1. W digrafie G wierzchołki nr 3 i 4 są incydentne tylko z jedną wspólną krawędzią: $(4, 3)$. Dlatego w 4-tym wierszu listy wymieniony jest wierzchołek 3 (a nie odwrotnie).

[1] W razie jakichkolwiek uwag do niniejszego dokumentu (choćby literówek) proszę o kontakt: `Elzbieta.Strzalka@fis.agh.edu.pl`

(b) **Macierz sąsiedztwa** również jest generowana bardzo podobnie do poznanych wcześniej przypadków: zakładając, że $a_{i,j}$ jest elementem macierzy sąsiedztwa w i-tym wierszu oraz j-tej kolumnie, dla digrafu prostego mamy zależność:

$$a_{i,j} = 1 \quad \Leftrightarrow \quad \text{w digrafie prostym } \textbf{istnieje} \text{ łuk } (i,j)$$
$$a_{i,j} = 0 \quad \Leftrightarrow \quad \text{w digrafie prostym } \textbf{nie istnieje} \text{ łuk } (i,j)$$

Z powyższej zależności wynika, że – w przeciwieństwie do przypadku grafów nieskierowanych – **macierz sąsiedztwa digrafów nie musi być symetryczna** względem diagonali. Macierz dla omawianego przykładowego digrafu przedstawiono na listingu 2.

```
0 0 0 0 0 0 1
1 0 1 0 0 1 1
0 1 0 0 0 1 0
0 0 1 0 1 0 0
0 0 1 0 0 0 0
0 0 0 0 1 0 0
1 0 0 0 0 0 0
```

Listing 2: **Macierz sąsiedztwa** grafu skierowanego z rys. 1.

(c) **Macierz incydencji**: każda kolumna macierzy koduje łuk (u, v) w taki sposób, że źródło u jest oznaczone wartością -1, a cel v: 1. Jako **źródło** rozumiemy wierzchołek, z którego wychodzi **krawędź prowadząca do wierzchołka** będącego **celem**. Macierz ma tyle kolumn, ile w digrafie znajduje się łuków. Przykład: listing 3.

```
-1   1   0   0   0   0   0   0   0   0   0   1
 0  -1  -1  -1  -1   1   0   0   0   0   0   0
 0   0   1   0   0  -1  -1   1   0   1   0   0
 0   0   0   0   0   0   0  -1  -1   0   0   0
 0   0   0   0   0   0   0   0   1  -1   1   0
 0   0   0   1   0   0   1   0   0   0  -1   0
 1   0   0   0   1   0   0   0   0   0   0  -1
```

Listing 3: **Macierz incydencji** grafu skierowanego z rys. 1. Łuk $(4, 3)$ jest zakodowany w ósmej kolumnie.

Model $G(n, p)$, który ma zostać wykorzystany do generowania losowych grafów, rozumiemy tak samo, jak w 1. projekcie: program powinien losować digraf o n wierzchołkach, gdzie p oznacza prawdopodobieństwo, że pomiędzy dowolnymi dwoma wierzchołkami istnieje łuk.

Ad. 2. Algorytm Kosaraju (*Algorytm Kosaraju oraz algorytmy z dalszych zadań zostały udostępnione na UPeL-u w osobnym pliku: "Algorytmy do projektu nr 4"*)

Algorytm Kosaraju służy do oznaczania **silnie spójnych składowych digrafu**. Digraf jest **silnie spójny**, jeśli dla dowolnych dwóch wierzchołków u i v istnieje droga z u do v. Wierzchołki digrafu, który nie jest ściśle spójny, można podzielić na **ściśle spójne składowe**, które same w sobie są ściśle spójne: **ściśle spójna składowa** to maksymalny zbiór wierzchołków digrafu

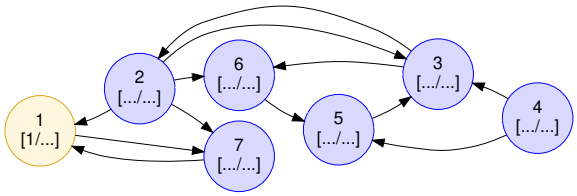
taki, że dla każdej pary wierzchołków u i v, należących do tej składowej, istnieje droga prowadząca z u do v (oraz z v do u, skoro wierzchołki mogą być wybrane w dowolny sposób).

Na tym etapie proszę otworzyć algorytm Kosaraju w pliku udostępnionym na UPeL-u pod nazwą: "Algorytmy do projektu nr 4".

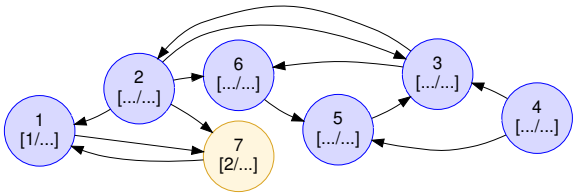
Algorytm Kosaraju polega na dwóch odrębnych **przeszukiwaniach** grafu wejściowego **w głąb**. Rozpoczynamy od oznaczenia wszystkich wierzchołków jako **nieodwiedzonych**. Potrzebne będą dwie tablice atrybutów:

- d – tablica **czasów odwiedzenia**; $d[v]$ oznacza czas odwiedzenia wierzchołka o numerze v;
- f – tablica **czasów przetworzenia**; $f[v]$ oznacza czas zakończenia przetwarzania wierzchołka o numerze v.

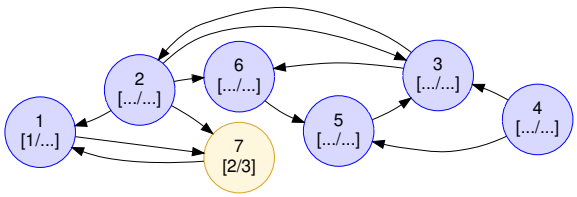
Startujemy z początkowym czasem $t = 0$. W **pierwszym przeszukiwaniu w głąb** wypełniamy tablice d oraz f: $d[v]$ jest uzupełniane aktualnym czasem t przy **odwiedzeniu** wierzchołka v, po czym rekurencyjnie odwiedzamy w głąb jego **nieodwiedzonych sąsiadów**, do których istnieje łuk rozpoczynający się od v. Natomiast $f[v]$ zostanie wypełnione czasem t wtedy, gdy w wyniku powrotów po zakończeniu rekurencji znowu dotrzemy do wierzchołka v, który nie ma już nieodwiedzonych sąsiadów. Po każdej operacji przypisania czasu t czas zostaje zwiększony.



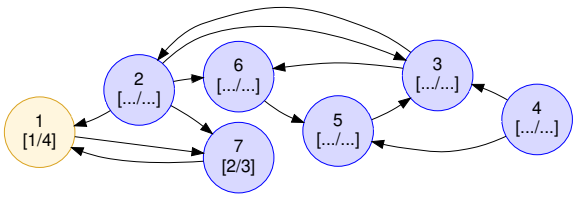
(a) **1. krok**: startujemy od dowolnego wierzchołka (tu: 1). Zapisujemy jego czas odwiedzenia.



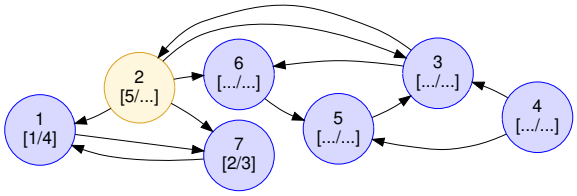
(b) **2. krok**: odwiedzenie sąsiada, którym jest wierzchołek 7.



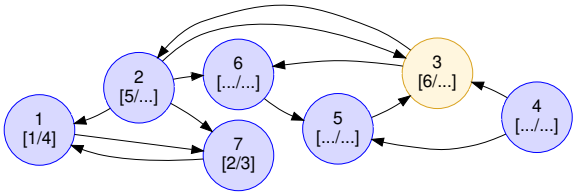
(c) **3. krok**: z wierzchołka nr 7 nie da się przejść do innego nieodwiedzonego wierzchołka; zapisujemy czas przetworzenia dla 7.



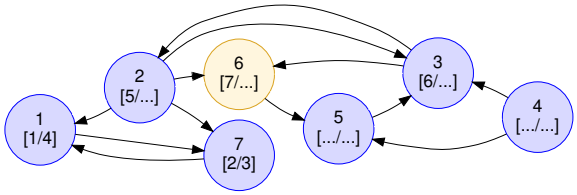
(d) **4. krok**: powrót do wierzchołka 1.; brak łuków do nieodwiedzonych wierzchołków $\Rightarrow$ zapisujemy czas przetworzenia dla 1.



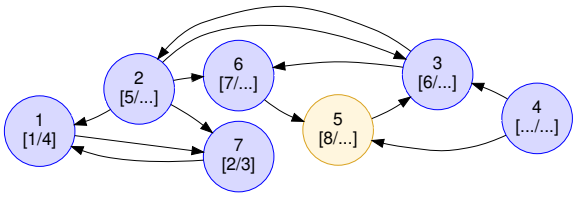
(e) **5. krok**: odwiedzamy nowy wierzchołek (nr 2).



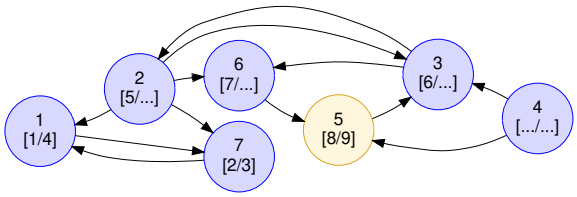
(f) **6. krok**: odwiedzamy sąsiada: nr 3.



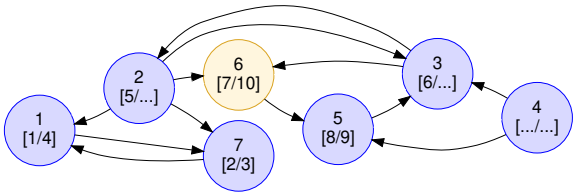
(g) **7. krok**: odwiedzamy sąsiada: nr 6.



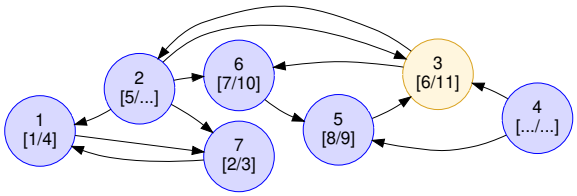
(h) **8. krok**: z 6. przechodzimy do wierzchołka nr 5.



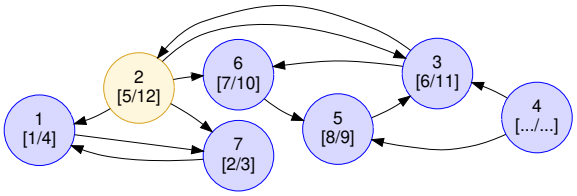
(i) **9. krok**: brak możliwości dalszego odwiedzania: oznaczamy czas przetworzenia dla 5. wierzchołka.



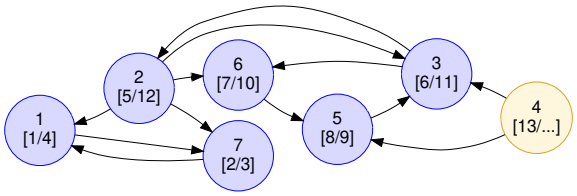
(j) **10. krok**: powrót do wierzchołka 6 i również brak możliwości dalszego odwiedzania: oznaczamy czas przetworzenia dla 6.



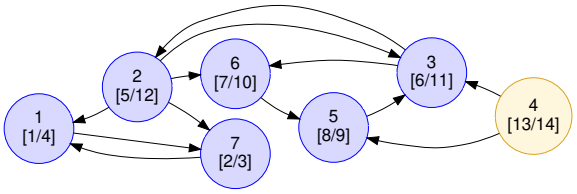
(k) **11. krok**: jak wyżej – zapisujemy czas przetworzenia wierzchołka nr 3.



(l) **12. krok**: powrót do wierzchołka 2., dla którego zapisujemy czas przetworzenia ze względu na brak nieodwiedzonych sąsiadów.



(m) **13. krok**: odwiedzamy nowy wierzchołek: nr 4.

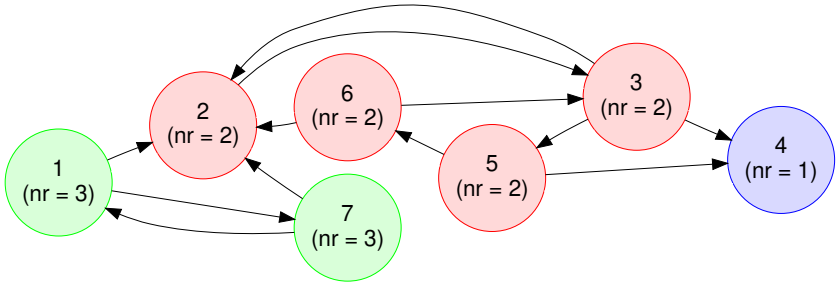


(n) **14. krok**: nie ma możliwości dalszego odwiedzania ⇒ zapisujemy czas przetworzenia wierzchołka nr 4, koniec pierwszego przeszukiwania w głąb.

Rysunek 2: Kolejne kroki pierwszego przeszukiwania w głąb w algorytmie Kosaraju dla grafu z rys. 1. W podpisie każdego wierzchołka v w nawiasie kwadratowym zapisano czas odwiedzenia oraz przetworzenia: $[d[v]/f[v]]$. Wierzchołek, który jest rozpatrywany w danym kroku, zaznaczono na żółto.

W dalszej części algorytmu będziemy pracować na **transpozycji** grafu G. **Transpozycją grafu** G, oznaczaną przez G^{T}, nazywamy graf utworzony przez **odwrócenie wszystkich łuków** oryginalnego grafu. Na transpozycji zostanie wykonane **drugie przeszukiwanie w głąb**, w którym nastąpi docelowe oznaczanie wierzchołków numerami silnie spójnych składowych. Zewnętrzna pętla tego przeszukiwania jest wykonywana **w kolejności malejących czasów przetworzenia** $f[v]$, tj. dla grafu z rys. 1. – kolejno: 4, 2, 3, 6, 5, 1, 7. Poszczególne kroki przeszukiwania

omówiono w podpisie do wykresu 3., który prezentuje wynik oznaczania silnie spójnych składowych przy pomocy algorytmu Kosaraju.



Rysunek 3: **Graf transponowany** G^{T} dla grafu z rys. 1. po drugim przeszukiwaniu w głąb algorytmu Kosaraju. Numer w nawiasie oznacza numer ściśle spójnej składowej. Wierzchołki są odwiedzane w kolejności: 4, 2, 3, 6, 5, 1, 7 (patrz: czasy przetworzenia na rys. 2(n)).

Działanie drugiego przeszukiwania w głąb: odwiedzamy wierzchołek nr 4, nadając mu numer składowej: 1. Z tego wierzchołka nie da się przejść dalej, więc zwiększamy numer składowej (teraz: $nr = 2$) i oznaczamy nim następny wierzchołek w zadanej przez czasy przetworzenia kolejności: wierzchołek nr 2. Odwiedzamy w głąb sąsiadów – odpowiednio: wierzchołki 3, 5 i 6, którym przypisujemy ten sam numer spójnej składowej. Wierzchołki nie mają już nieodwiedzonych sąsiadów, więc wracamy na najwyższy poziom rekurencji. Zwiększamy numer spójnej składowej (wynosi już 3) i oznaczamy nim następny nieodwiedzony wierzchołek według kolejności malejących czasów przetworzenia: będzie to wierzchołek nr 1. Z niego przechodzimy już tylko do wierzchołka nr 7, który należy do tej samej składowej.

***Implementacja ze stosem**: w alternatywnej wersji algorytmu Kosaraju w pierwszym przeszukiwaniu w głąb zapisywanie czasu przetworzenia zastępuje się odkładaniem wierzchołka na stos po zakończeniu rekurencji. Wówczas w drugim przeszukiwaniu w głąb pobiera się ze stosu wierzchołek i – jeśli jest nieodwiedzony – w grafie transponowanym uruchamiamy przeszukiwanie w głąb zaczynając od tego wierzchołka i oznaczając składową. Po powrocie na najwyższy poziom rekurencji pobieramy ze stosu kolejny wierzchołek i wykonujemy te same operacje, aż stos będzie pusty.*

Ad. 3. Algorytm Bellmana-Forda

Na podstawie zaimplementowanych wcześniej algorytmów należy wygenerować **silnie spójny losowy digraf** oraz przypisać jego krawędziom losowe wagi z przedziału $[-5, 10]$.

Algorytm **Bellmana-Forda** pozwala na znalezienie najkrótszych ścieżek z wierzchołka startowego s dla grafu o wagach, które mogą być ujemne – w przeciwieństwie do algorytmu Dijkstry, który może być używany tylko przy nieujemnych wagach. Oba algorytmy zaczynamy podobnie: od **inicjalizacji tablic atrybutów** d_s i p_s dla każdego wierzchołka (algorytm pomocniczy $\textsc{Init}(G, s)$ z poprzedniego zestawu).

Następnie w algorytmie Bellmana-Forda wykonujemy **relaksację każdej krawędzi w dowolnej kolejności**. Przejście po wszystkich krawędziach powtarzamy dokładnie $n-1$ razy (n jest liczbą wierzchołków grafu G). Jest to liczba iteracji relaksacji wszystkich krawędzi, która zapewnia, że

wszystkie najkrótsze ścieżki zostaną już znalezione. Nie jest to jednak koniec algorytmu: wykonujemy **jeszcze jedną pętlę po wszystkich krawędziach**, w której sprawdzamy, czy ewentualna dalsza relaksacja spowodowałaby dalsze skrócenie którejkolwiek ze ścieżek. Jeśli tak, to w grafie G istnieje **cykl o ujemnej wadze** osiągalny z wierzchołka startowego s, który **uniemożliwia określenie najkrótszych ścieżek** od tego wierzchołka – algorytm kończy się zgłoszeniem informacji o błędzie.

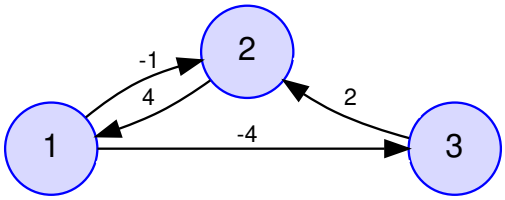
Algorytm Bellmana-Forda wykonuje dużo więcej relaksacji, niż algorytm Dijkstry, ponieważ nie optymalizuje procesu wyboru kolejności relaksacji krawędzi. Ma jednak dwie istotne zalety: zdolność identyfikowania problemu cyklu o ujemnej wadze oraz możliwość działania dla grafu o ujemnych wagach.

Wynik działania algorytmu Bellmana-Forda zostanie przedstawiony na przykładzie 4. zadania, ponieważ jest częścią składową omawianego tam algorytmu Johnsona.

Ad. 4. Algorytm Johnsona

Na tym etapie proszę przyjrzeć się algorytmowi Johnsona udostępnionemu w pliku "Algorytmy do projektu nr 4".

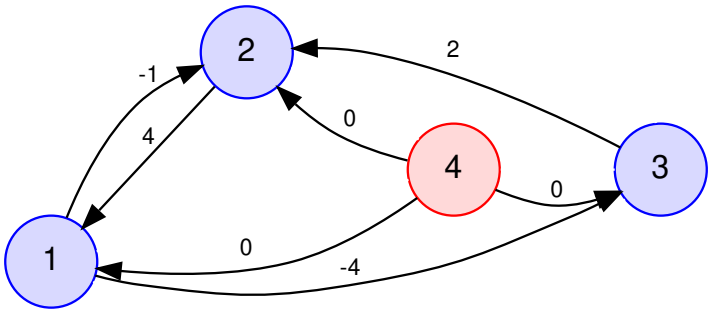
Algorytm Johnsona pozwala na określenie długości najkrótszych ścieżek pomiędzy wszystkimi parami wierzchołków: dzięki temu otrzymamy **macierz odległości**. Działanie algorytmu zostanie przedstawione na przykładowym digrafie G , który jest silnie spójny, zaprezentowanym na rys. 4.



Rysunek 4: Przykładowy losowy digraf G

Algorytm zaczyna się od sprawdzenia, czy w grafie wejściowym G istnieje **cykl o ujemnej wadzie** osiągalny z któregokolwiek z wierzchołków: jeśli tak, to algorytm Johnsona musi zakończyć działanie ze względu na niepowodzenie. W tym celu tworzymy graf G' poprzez dodanie do grafu G **nowego wierzchołka** s. Nowy wierzchołek łączymy z wszystkimi pozostałymi wierzchołkami **krawędziami skierowanymi od** s, którym przypisujemy wagę 0 (rys. 5). Dla tak powstałego grafu G' wywołujemy algorytm **Bellmana-Forda**, przekazując nowy wierzchołek s jako wierzchołek startowy. Jeśli w wejściowym grafie G istnieje cykl o ujemnej sumie wag, to z całą pewnością jest on osiągalny ze źródła s, więc algorytm Bellmana-Forda nas o tym poinformuje.

Dzięki algorytmowi Bellmana-Forda uzyskujemy tablicę długości najkrótszych ścieżek od wierzchołka s: d_s (o ile algorytm nie wykrył cyklu o ujemnej wadze). Dla grafu G' z rys. 5. po trzech iteracjach relaksacji wszystkich krawędzi algorytm Bellmana-Forda obliczył wyniki przedstawione w tabeli 1.



Rysunek 5: Digraf $G' = G \cup s$, powstały przez dodanie do grafu G nowego wierzchołka nr 4 wraz z krawędziami do pozostałych wierzchołków: $(4,1)$, $(4,2)$ oraz $(4,3)$. Nowe krawędzie mają zerową wagę.

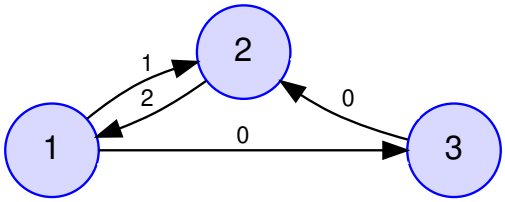
$d_4(1) = 0$	$p_4(1) = 4$
$d_4(2) = -2$	$p_4(2) = 3$
$d_4(3) = -4$	$p_4(3) = 1$
$d_4(4) = 0$	———

Tabela 1: Długości ($\equiv$ wagi) najkrótszych ścieżek z wierzchołka startowego $s = 4$ dla grafu G' z rys. 5 do pozostałych wierzchołków, uzyskane przy pomocy algorytmu Bellmana-Forda, wraz z tabelą poprzedników.

Długości d_4 dla każdego wierzchołka zachowujemy w tablicy h, która posłuży do **przeskalowania wag krawędzi grafu** G w taki sposób, żeby były **nieujemne** – dla każdej krawędzi (u, v) obliczamy nową wagę:

$$\widehat{w}(u,v) = w(u,v) + h(u) - h(v). \tag{1}$$

Dowodzi się, że takie przeskalowanie nie powoduje zmiany najkrótszych ścieżek (ma wpływ tylko na ich długość). Graf G z nowymi wagami $\widehat{w}(u,v)$ przedstawiono na rys. 6.



Rysunek 6: Graf G po przeskalowaniu wag zgodnie z wzorem (1) – por. rys. 4.

Dla tak powstałego digrafu można już wywołać algorytm Dijkstry, skoro pozbyliśmy się ujemnych wag. Dlatego kolejnym krokiem algorytmu Johnsona jest wielokrotne wywołanie **algorytmu Dijkstry**, żeby uzyskać macierz odległości (analogicznie do 3. zadania w poprzednim zestawie).

Należy jednak pamiętać o **powrotnym przeskalowaniu wag**:

$$D_{u,v} = \widehat{d}_u(v) - h(u) + h(v), \tag{2}$$

gdzie:

- u – wierzchołek startowy aktualnego wywołania algorytmu Dijkstry;
- $\widehat{d}_u(v)$ – długość najkrótszej ścieżki z wierzchołka u do wierzchołka v, znaleziona dla grafu o zmodyfikowanych, nieujemnych wagach;
- $D_{u,v}$ – długość najkrótszej ścieżki z wierzchołka u do wierzchołka v w oryginalnym grafie G.

W wyniku wywołań algorytmu Dijsktry dla każdego z wierzchołków grafu z rys. 6 jako źródła s otrzymano długości najkrótszych ścieżek $\widehat{d}_s$, zaprezentowane w tabeli 2.

$\widehat{d}_1(1) = 0$	$\widehat{d}_2(1) = 2$	$\widehat{d}_3(1) = 2$
$\widehat{d}_1(2) = 0$	$\widehat{d}_2(2) = 0$	$\widehat{d}_3(2) = 0$
$\widehat{d}_1(3) = 0$	$\widehat{d}_2(3) = 2$	$\widehat{d}_3(3) = 0$

(a) Wierzchołek startowy: $s = 1$. (b) Wierzchołek startowy: $s = 2$. (c) Wierzchołek startowy: $s = 3$.

Tabela 2: Wyniki wszystkich wywołań algorytmu Dijkstry dla grafu G ze zmodyfikowanymi wagami (jak na rys. 6).

Po powrotnym przeskalowaniu wyników zgodnie z wzorem (2) otrzymano ostateczne wartości elementów **macierzy odległości**, zaprezentowane na listingu 4.

```
0   -2   -4
4    0    0
6    2    0
```

Listing 4: **Macierz odległości** grafu skierowanego z rys. 4, uzyskana przy pomocy algorytmu Johnsona.

Dzięki opisanej procedurze można skorzystać z bardziej optymalnego algorytmu Dijkstry w zamian za wielokrotne wywołania algorytmu Bellmana-Forda. Jednokrotne wywołanie algorytmu Bellmana-Forda jest jednak konieczne do skontrolowania, czy w grafie nie ma cyklu o ujemnej wadze, oraz do uzyskania parametrów h pozwalających na przeskalowanie wag do nieujemnych wartości.