TEORIA GRAFÓW

Podstawowe definicje:

- **Graf (nieskierowany)** $G = (V, E)$ – struktura składająca się ze:
  - zbioru **wierzchołków** $V = \{v_1, v_2, \ldots, v_n\}$ oraz
  - zbioru **krawędzi** $E = \{e_1, e_2, \ldots, e_m\}$.

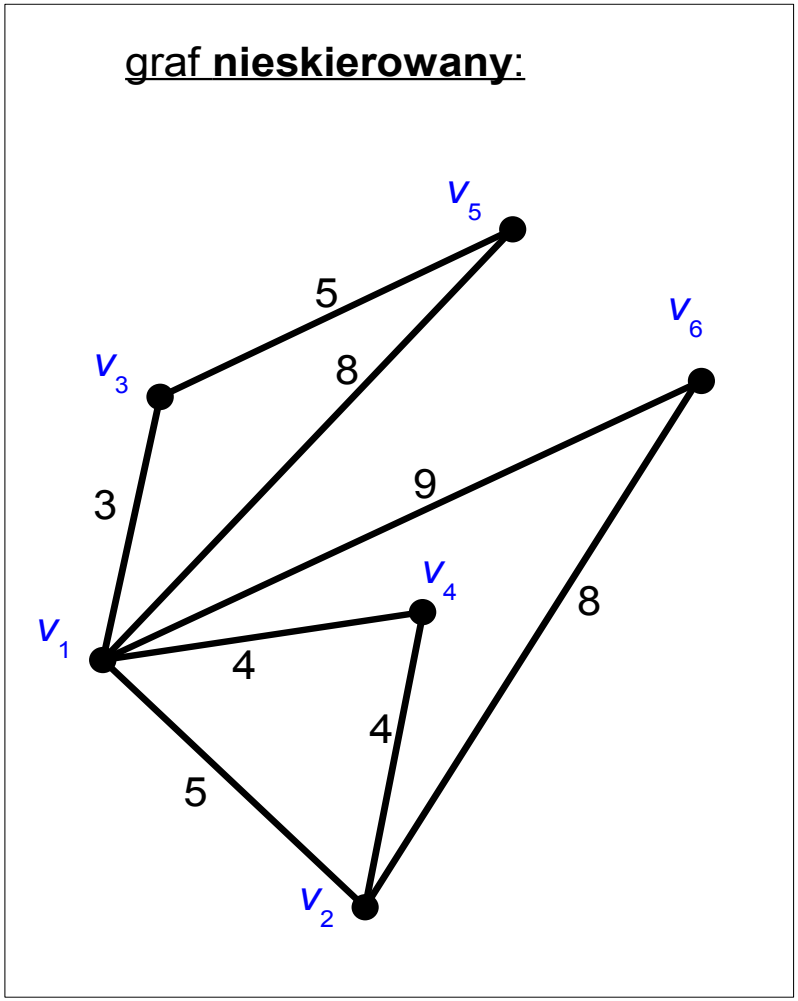
  Z każdą krawędzią e skojarzona jest para wierzchołków *incydentnych* (u, v). Analizę algorytmów grafowych możemy ograniczyć do przypadków, w których para wierzchołków jednoznacznie identyfikuje krawędź.

- **Graf skierowany** – dla każdej krawędzi (oznaczanej tutaj jako **łuk**) para wierzchołków incydentnych jest **parą uporządkowaną** $\{u, v\}$.
  Zatem w grafie skierowanym $\{u, v\} \neq \{v, u\}$ ($\{u, v\}$ i $\{v, u\}$ oznacza dwa różne łuki, podczas gdy w grafie nieskierowanym (u, v) i (v, u) to ta sama krawędź, tzn. $(u, v) = (v, u)$).

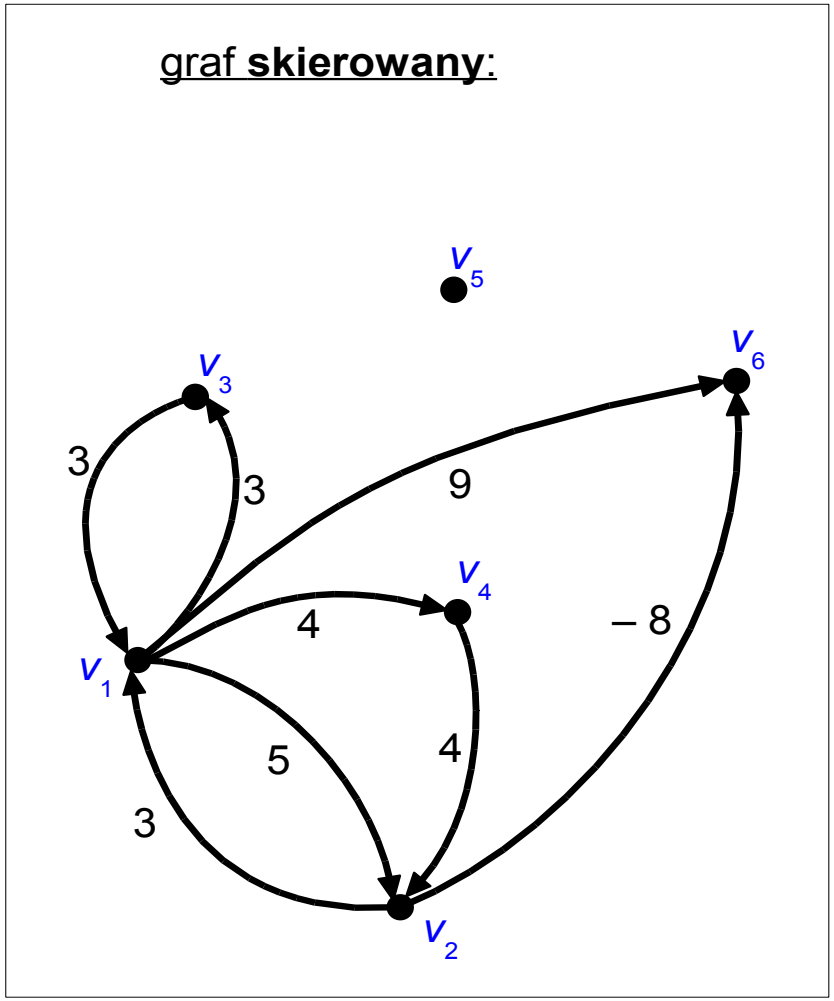
- Każdej krawędzi (łukowi) będziemy przypisywać **wagę** (liczbę), reprezentującą (w zastosowaniach praktycznych) np. długość drogi, koszt budowy drogi, czas lub koszt przejazdu, niezawodność połączenia, prawdopodobieństwo przejścia, przepustowość.
- **Rozmiar grafu** będziemy identyfikować przez **liczbę wierzchołków** n oraz liczbę **krawędzi** (łuków) m.

Przykłady **grafów** o rozmiarze ***n* = 6** i ***m* = 8**

graf **nieskierowany**:



graf **skierowany**:



Struktury grafowe

Macierz wag – tablica 2-wymiarowa o rozmiarze $n \times n$, w której wartość na przecięciu i-tego wiersza i j-tej kolumny oznacza wagę krawędzi (i, j) (łuku $\{i, j\}$).

Jeśli dany graf nie zawiera krawędzi (i, j) to w odpowiadającej jej komórce macierzy wpisujemy wartość ∞. Wartości wyróżnione (zera, wartości ujemne, nieskończoności, znaki nienumeryczne) znajdują się też na przekątnej macierzy (brak krawędzi typu (i, i)).

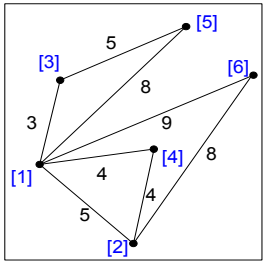
Zauważmy również, że macierz dla grafu nieskierowanego jest zawsze symetryczna (w takim wypadku można również przechowywać tylko połowę macierzy).

Zajętość pamięci – $\boldsymbol{O(n^2)}$.

Lista krawędzi – tablica 2-wymiarowa o rozmiarze $m \times 3$ (lub 3 tablice liniowe o długości m), gdzie pierwsza kolumna (tablica) zawiera wierzchołki początkowe, druga kolumna (tablica) – wierzchołki końcowe, a trzecia kolumna (tablica) – wagi wszystkich krawędzi grafu. Krawędzie nieistniejące nie muszą być tutaj przechowywane.

Zajętość pamięci – $\boldsymbol{O(m)}$.

Przykłady **struktur grafowych**



Macierz wag

	[1]	[2]	[3]	[4]	[5]	[6]
[1]	–	5	3	4	8	9
[2]	5	–	∞	4	∞	8
[3]	3	∞	–	∞	5	∞
[4]	4	4	∞	–	∞	∞
[5]	8	∞	5	∞	–	∞
[6]	9	8	∞	∞	∞	–

Lista krawędzi

	A	B	W
(1)	[1]	[2]	5
(2)	[1]	[3]	3
(3)	[1]	[4]	4
(4)	[1]	[5]	8
(5)	[1]	[6]	9
(6)	[2]	[4]	4
(7)	[2]	[6]	8
(8)	[3]	[5]	5

Połączone listy sąsiadów – struktura składająca się z tablicy n list, gdzie lista dowiązana do komórki i-tej zawiera wszystkie krawędzie incydentne z wierzchołkiem i-tym (waga krawędzi i oznaczenie wierzchołka).

Zauważmy, że w przypadku grafu nieskierowanego każda krawędź (i, j) występuje w tej strukturze 2 razy – w liście i-tej oraz j-tej.

Zaletą struktury jest to, że wszystkie krawędzie (łuki) z danego wierzchołka mamy zgrupowane w jednym miejscu (są dostępne bez konieczności przeszukiwania całego grafu).

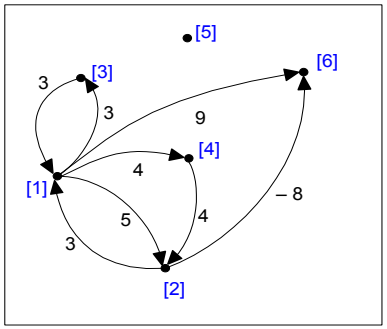
Zajętość pamięci – $\boldsymbol{O(n+m)}$.

Pęk wyjściowy – odmiana poprzedniej struktury dla sytuacji, gdy nie wykonujemy na grafie operacji dodawania / usuwania krawędzi. Składa się z 3 tablic – pierwszej o długości n oraz dwóch o długości m.

Dwie ostatnie tablice zawierają odpowiednio etykiety wierzchołków końcowych i wagi wszystkich krawędzi, pogrupowane w zbiory ze sobą incydentne, a pierwsza tablica (odpowiadająca wierzchołkom początkowym) zawiera wskaźniki rozdzielające poszczególne grupy (np. jeśli wierzchołek 1 ma 5 krawędzi incydentnych, to komórki [1] i [2] pierwszej tablicy zawierają odpowiednio wskaźniki na komórki [1] i [6] tablicy drugiej).

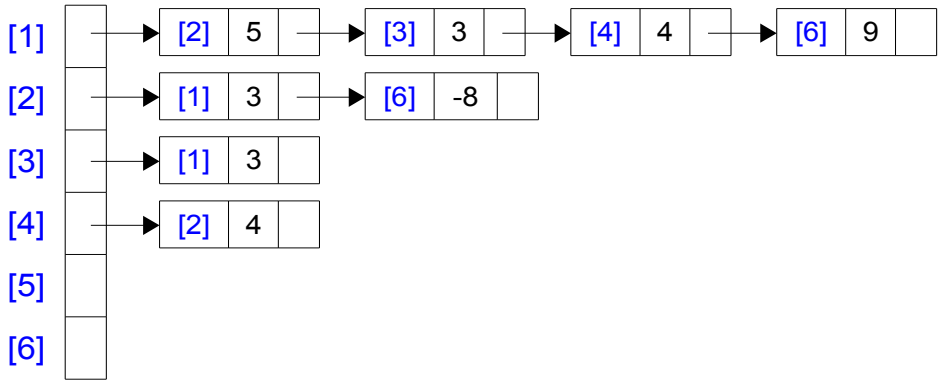
Struktura posiada zaletę struktury wcześniejszej – zgrupowanie krawędzi incydentnych poszczególnych wierzchołków, przy braku typów wskaźnikowych.

Zajętość pamięci – $\boldsymbol{O(n+m)}$.

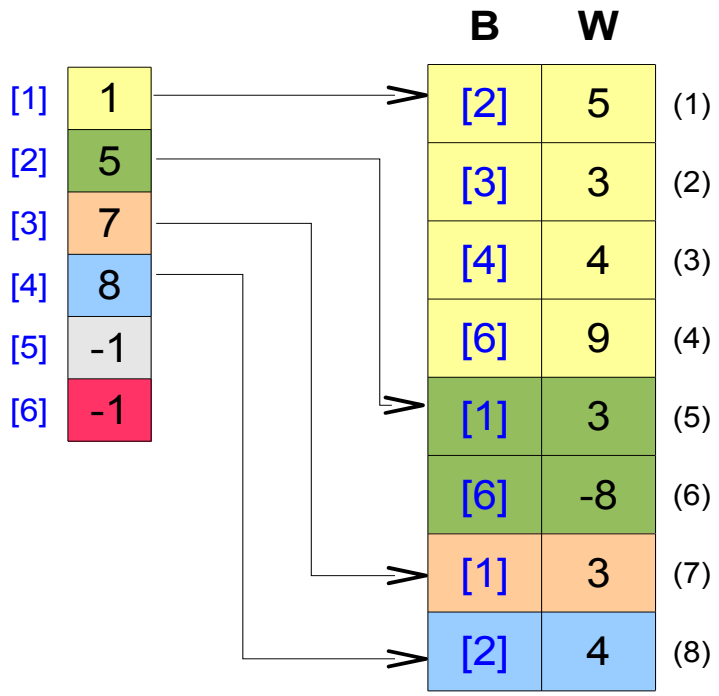


Przykłady **struktur grafowych** (c.d.)

Połączone listy sąsiadów



Pęk wyjściowy



Dalsze definicje

Droga (ścieżka) z wierzch. v_1 do v_k w grafie skierowanym G to ciąg łuków $\{\{v_1, v_2\}, \{v_2, v_3\}, \ldots, \{v_{k-1}, v_k\}\}$, który możemy jednoznacznie określić po prostu przez **ciąg (permutację) wierzchołków**, przez które droga ta przebiega, tj. $\{v_1, v_2, \ldots, v_{k-1}, v_k\}$.

Wagą drogi będziemy określać sumę wag łuków ją tworzących.

Cykl – droga zamknięta, tzn. taka, że $v_1 = v_k$.

Graf acykliczny – graf nie zawierający cykli.

Zauważmy, że z wierzchołka s do t może istnieć wiele różnych dróg, zatem **najkrótszą drogą** będzie ta spośród dróg, która ma najmniejszą wagę.

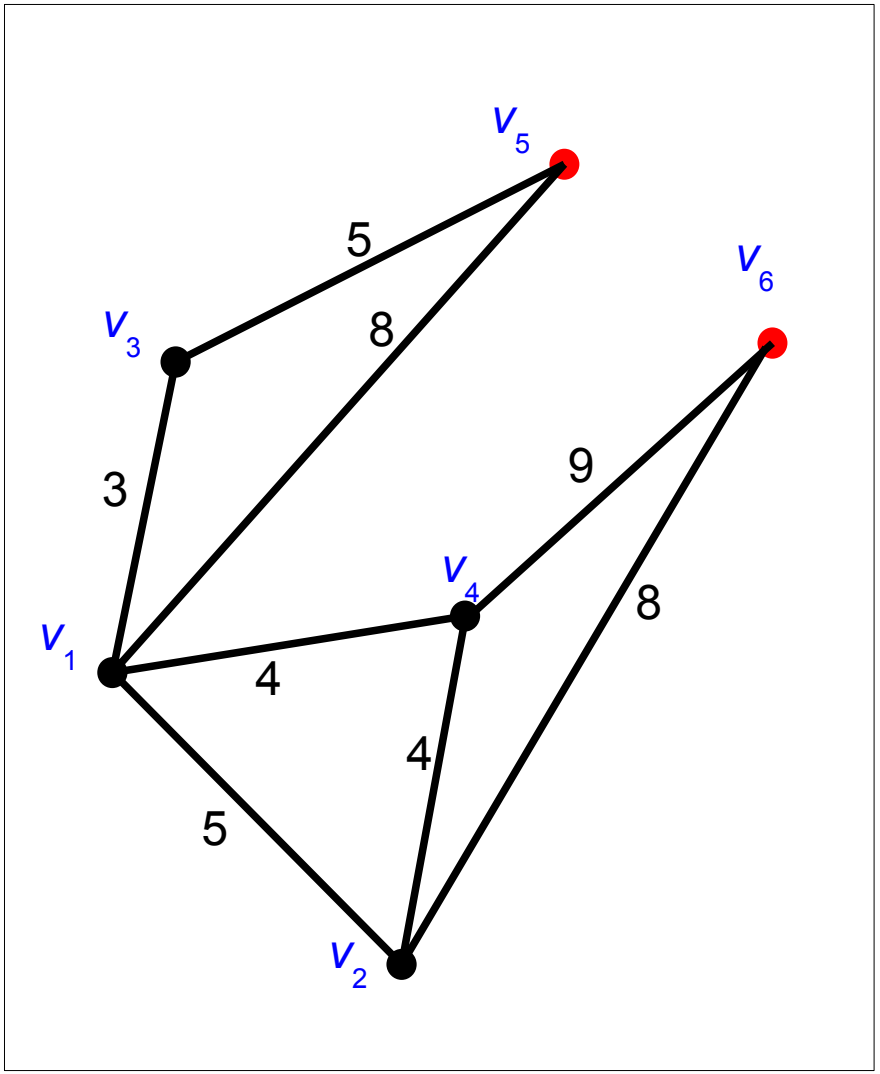
Podgraf $G' = (V', E')$ grafu $G = (V, E)$ to graf o rozmiarze $n' \leq n$ i $m' \leq m$, dla którego $V' \subseteq V$ oraz $E' \subseteq E$ (tzn. wszystkie wierzchołki i łuki G' należą do G).

Graf spójny (nieskierowany) – graf, dla którego istnieje droga między każdą parą wierzchołków. Spójny graf **skierowany** to taki, którego wersja nieskierowana jest spójna.

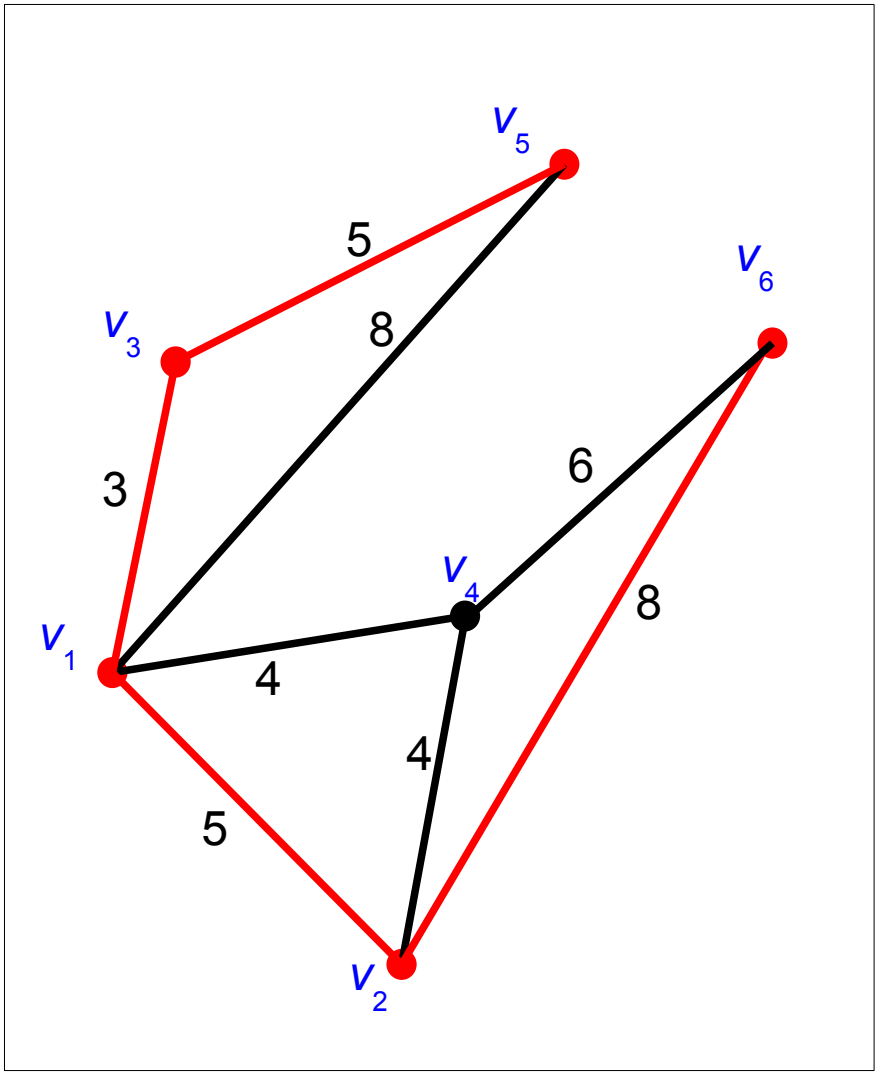
Drzewo – spójny, nieskierowany graf acykliczny. W drzewie, jak w każdym grafie spójnym (nieskierowanym), istnieje droga między każdą parą wierzchołków. Zatem dołączenie nowej krawędzi spowoduje powstanie cyklu, a usunięcie jakiejkolwiek krawędzi z drzewa spowoduje jego rozspójnienie. Wreszcie, w drzewie $m = n - 1$.

Drzewo rozpinające (spinające) $\boldsymbol{T} = (\boldsymbol{V^T}, \boldsymbol{E^T})$ – podgraf (nieskierowanego, spójnego) grafu $G = (V, E)$, który jest spójny i $V^T = V$. Graf spójny może zawierać wiele (do n^{n-2}) różnych drzew rozpinających, z których to o najmniejszej wadze nazywamy **Minimalnym Drzewem Rozpinającym** (Spinającym) – **MST** (ang. ***Minimum Spanning Tree***).

Przykłady **dróg** z wierzchołka v_5 **do** v_6



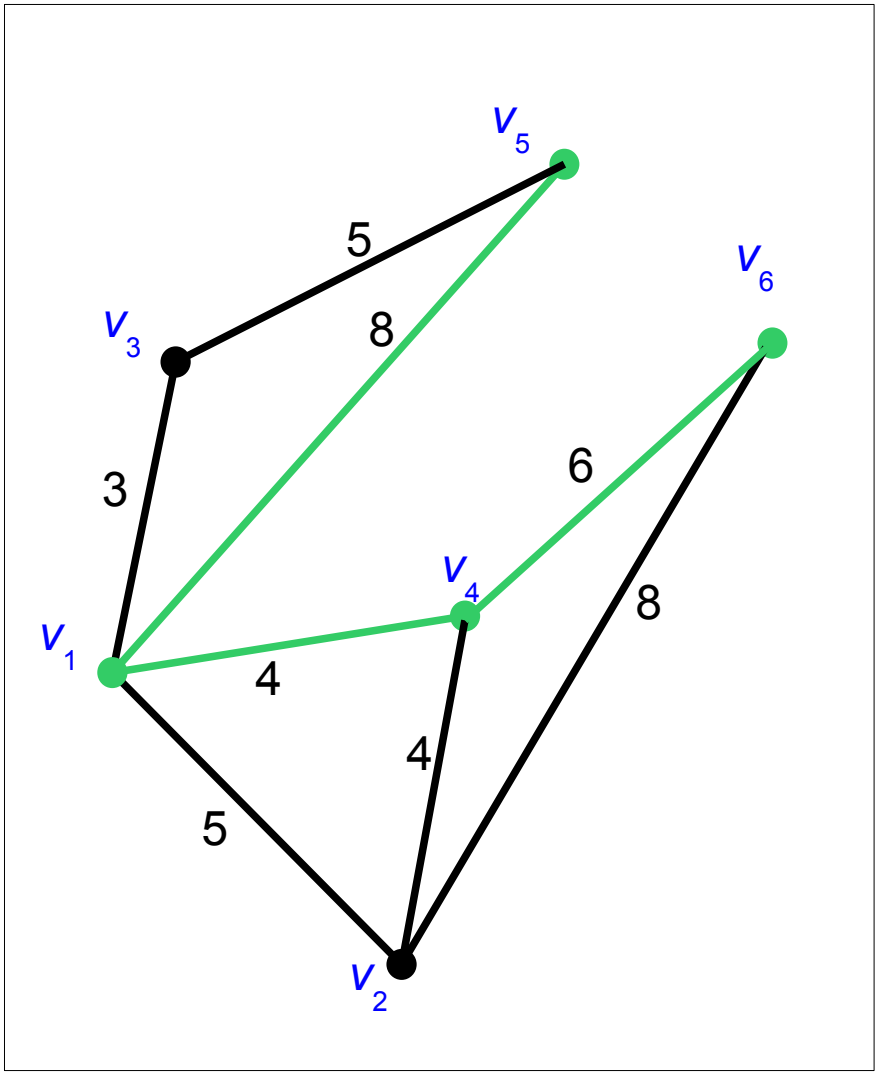
Przykłady **dróg** z wierzchołka v_5 **do** v_6



Droga D1 $\{v_5, v_3, v_1, v_2, v_6\}$

Waga D1: 5+3+5+8 **=** **21**

Przykłady **dróg** z wierzchołka v_5 **do** v_6



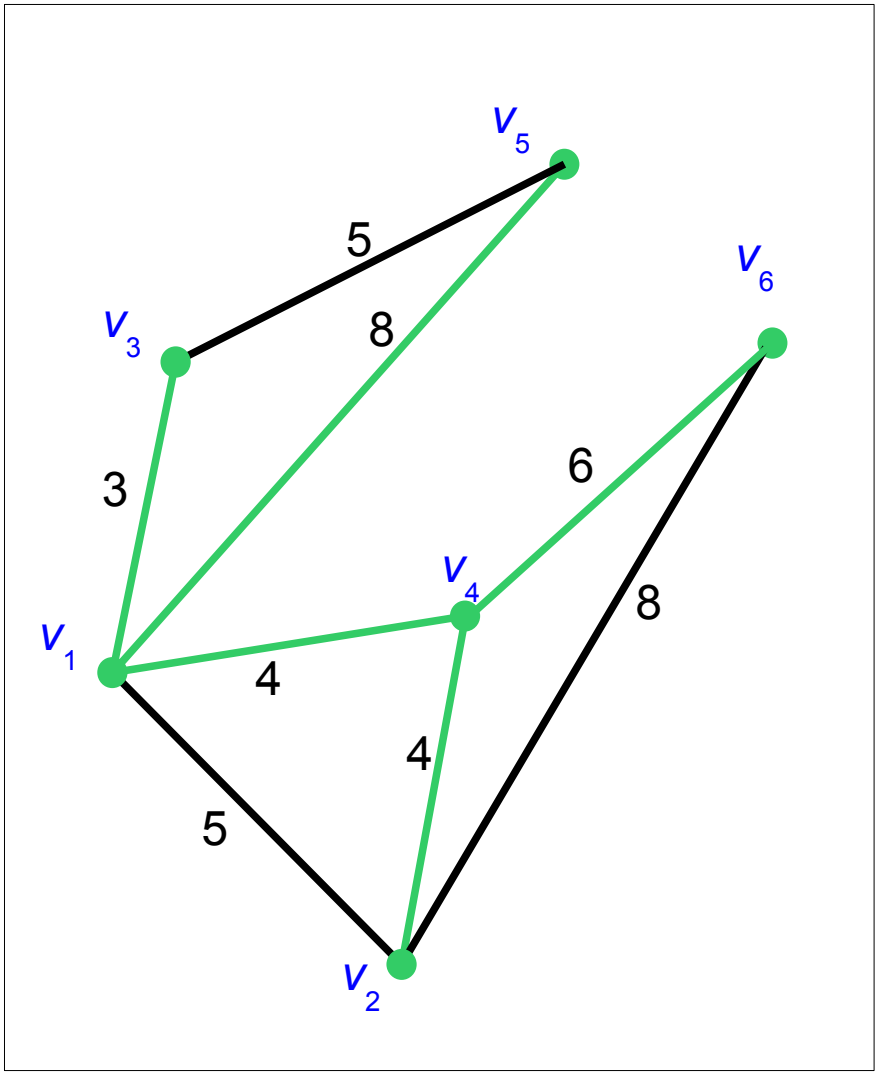
Droga D1 $\{v_5, v_3, v_1, v_2, v_6\}$

Waga D1: 5+3+5+8 **=** **21**

Droga D2 $\{v_5, v_1, v_4, v_6\}$

Waga D2: 8+4+6 **=** **18**

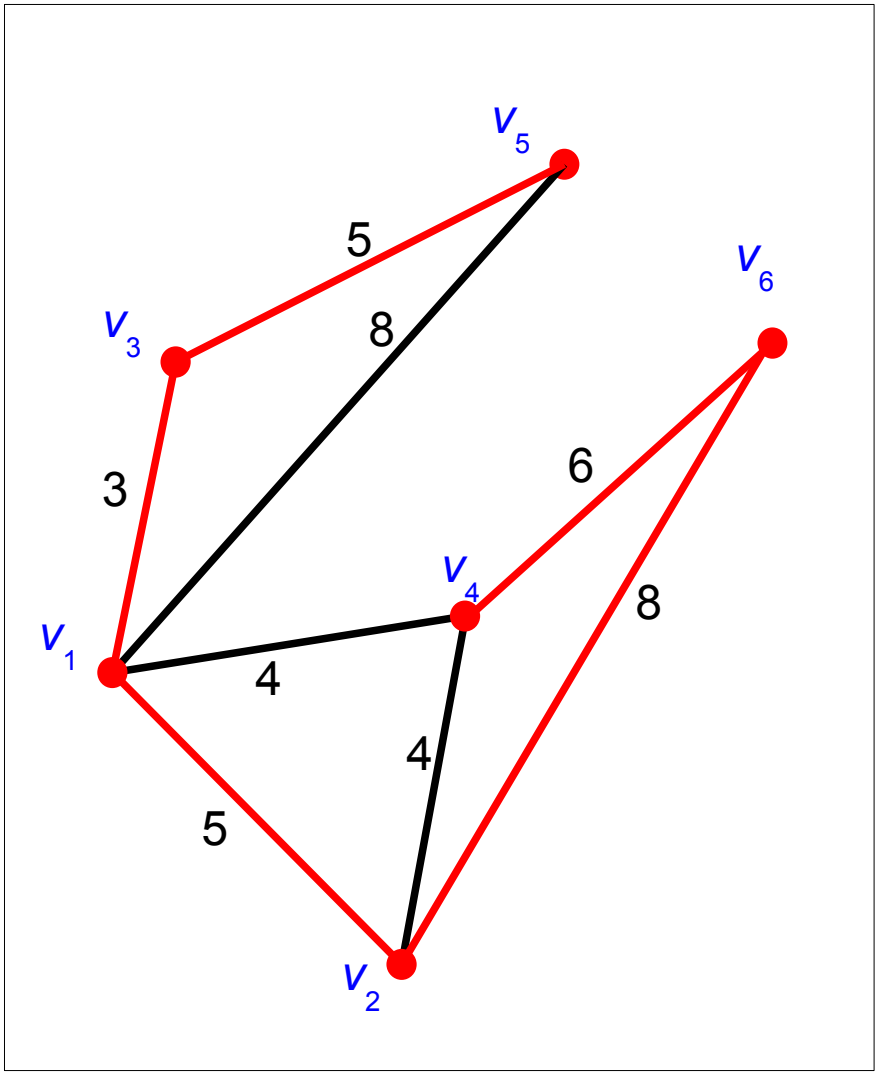
Przykłady drzew **MST**



Drzewo MST1

Waga MST1: 3+8+4+4+6 **= 25**

Przykłady drzew **MST**



Drzewo MST1

Waga MST1: 3+8+4+4+6 **=** **25**

Drzewo MST2

Waga MST2: 5+3+5+8+6 **=** **27**

Algorytmy wyznaczania MST

Algorytm KRUSKALA (1956)

polega na dołączaniu kolejno krawędzi w porządku ich niemalejących wag, o ile dołączana krawędź nie utworzy cyklu z już wybranymi krawędziami.

Aby wyznaczyć złożoność obliczeniową algorytmu, musimy znaleźć efektywną **metodę sprawdzania, czy dołączana krawędź utworzy cykl** z już dołączonymi.

Zauważmy, że dołączane **krawędzie nie muszą tworzyć spójnego grafu** z już dołączonymi (spójny graf na pewno otrzymujemy w pierwszym kroku – gdy mamy dołączoną tylko 1 krawędź – i potem możemy otrzymać dopiero w ostatnim kroku – gdy otrzymujemy pełne drzewo).

Zatem powyższą procedurę możemy skonstruować w ten sposób, że tworzymy **tablicę C o rozmiarze n** i przy jej pomocy kolorujemy wierzchołki.

Na początku każdej komórce przypisujemy inną wartość (reprezentującą kolor danego wierzchołka), np. $C[i] = i$ dla $i = 1, 2, \ldots, n$.

W trakcie działania procedury, wierzchołki należące do tego samego poddrzewa będą posiadać ten sam kolor.

Zatem **na początku** każdy wierzchołek należy do innego poddrzewa, a **na końcu** wszystkie wierzchołki muszą należeć do tego samego poddrzewa.

Gdy chcemy **dołączyć** powiedzmy **krawędź** (u, v), to musimy rozpatrzyć następujące przypadki:

1. Żaden wierzchołek (ani u, ani v) jeszcze nie został wybrany (tzn. nie dołączono jeszcze krawędzi, która byłaby incydentna z u lub z v).
   Zatem każdy z nich ma inny kolor (taki, którego nie posiada żaden inny wierzchołek).
   Wtedy wierzchołkowi u nadajemy kolor wierzch. v ($C[u] = C[v]$) lub odwrotnie ($C[v] = C[u]$) – oba wierzchołki otrzymują ten sam kolor, a więc należą od teraz do tego samego (nowego) poddrzewa. Taka operacja oczywiście **nie utworzy cyklu**, a więc jest dopuszczalna.

2. Jeden wierzch., np. u, był już wcześniej wybrany, a drugi, v, jeszcze nie.
   Zatem dołączenie krawędzi (u, v) **nie spowoduje powstania cyklu**.
   Wtedy wierzchołek jeszcze nie wybrany musimy dołączyć do poddrzewa, do którego należy wierzchołek u (tzn. wierzchołkowi v przypisujemy kolor wierzchołka u: $C[v] = C[u]$).

3. Oba wierzchołki już wcześniej zostały wybrane, z tym że należą do różnych poddrzew (wierzchołki u i v posiadają więc różne kolory, podobnie jak we wcześniejszych przypadkach).
   Zatem dołączenie krawędzi spowoduje połączenie dwóch różnych poddrzew, ale **nie spowoduje to powstania cyklu**. W tej sytuacji wszystkie wierzchołki jednego poddrzewa muszą przyjąć kolor drugiego, np. $C[i] = C[v]$ jeśli $C[i] == C[u]$.

4. Oba wierzchołki już wcześniej zostały wybrane i należą **do tego samego poddrzewa** (posiadają ten sam kolor).
   Wtedy próbujemy dołączyć nową krawędź do istniejącego drzewa, co – jak już wiadomo – zawsze **spowoduje powstanie cyklu**. Zatem taka operacja jest zabroniona.

Podsumowując powyższe, **dołączenie krawędzi (u, v) jest zabronione** (powstanie cykl) jeśli $\boldsymbol{C[u] == C[v]}$ (przypadek 4). W przeciwnym wypadku należy dołączyć krawędź i wykonać operacje na tablicy C opisane odpowiednio w punktach 1, 2 lub 3.

Oczywiście sprawdzenie powyższego warunku ($C[u] == C[v]$) zajmie $O(1)$ czasu, jednak przypadek 3 może w skrajnej sytuacji wymagać przekolorowania $n-2$ wierzchołków (przy wprowadzeniu dodatkowych zmiennych przechowujących informacje o rozmiarze poszczególnych poddrzew, operację tą można skrócić do $\lfloor n/2 \rfloor$ operacji). Zatem operacja sprawdzenia, czy dołączenie pojedynczej krawędzi do już dołączonych spowoduje powstanie cyklu zajmie $\boldsymbol{O(n)}$ **czasu**.

Teraz możemy wrócić do wyznaczenia złożoności całego algorytmu Kruskala. Zależy ona od kilku czynników.

Jeśli do przechowywania grafu wykorzystamy listę krawędzi i przed rozpoczęciem dołączania krawędzi posortujemy tą strukturę niemalejąco ze względu na wagi (co zajmie $O(m \log m)$ czasu) to będziemy mogli następnie **analizować krawędzie po kolei**, za każdym razem stosując procedurę sprawdzania cyklu.

Liczba takich iteracji, oznaczmy ją przez p, będzie się zawierała między $n-1$ (żadna z analizowanych krawędzi nie tworzyła cyklu) a m (dopiero ostatnia krawędź spowodowała dołączenie wszystkich wierzchołków i zespójnienie poddrzew). Zatem **złożoność** takiej wersji **algorytmu Kruskala** wyniesie $\boldsymbol{O(m \log m + pm)}$.

Zauważmy jednak, że w przypadku grafów gęstych ($m >> n$) tylko niewielka część wszystkich krawędzi będzie analizowana. Wtedy sortowanie wszystkich będzie niepotrzebne – zbyt rozrzutne.

Lepiej wtedy **umieścić krawędzie w kopcu** i pobierać za każdym razem najkrótszą z jeszcze nie dołączonych z korzenia.

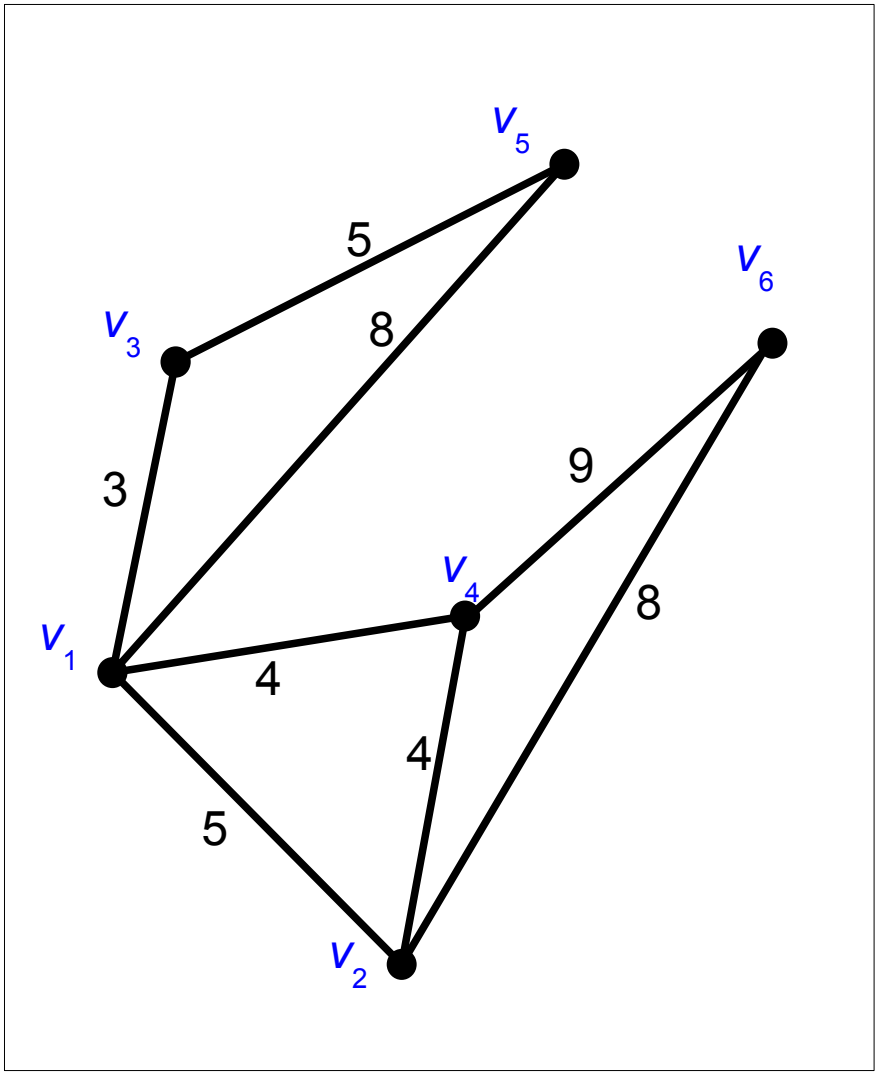
Wtedy takie pobranie będzie zajmować $O(\log m)$, a takich pobrań będzie p, zatem złożoność całego algorytmu wyniesie

$$\boldsymbol{O(p(\log m + m)) = O(pm)},$$

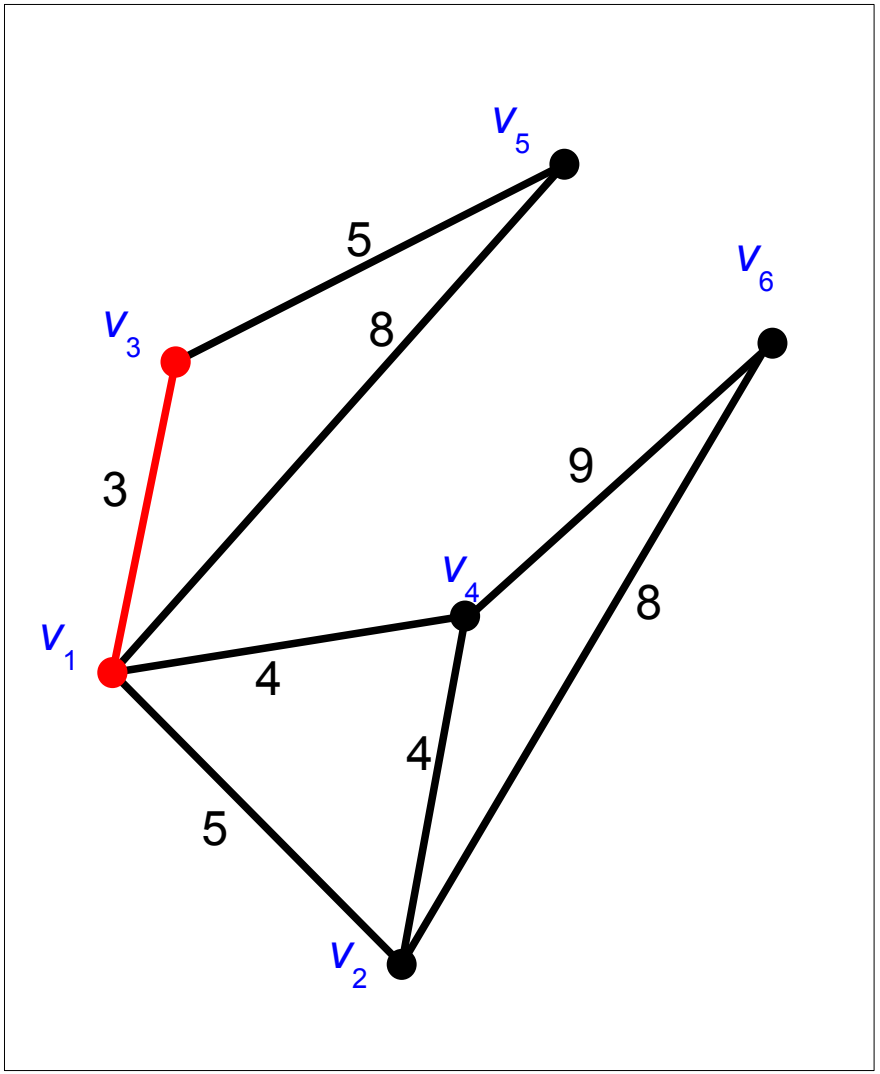
gdzie $n \leq p \leq m$.

Możliwe jest też takie zaimplementowanie algorytmu Kruskala, że jego złożoność wyniesie $\boldsymbol{O(m \log m)}$.

Algorytm KRUSKALA

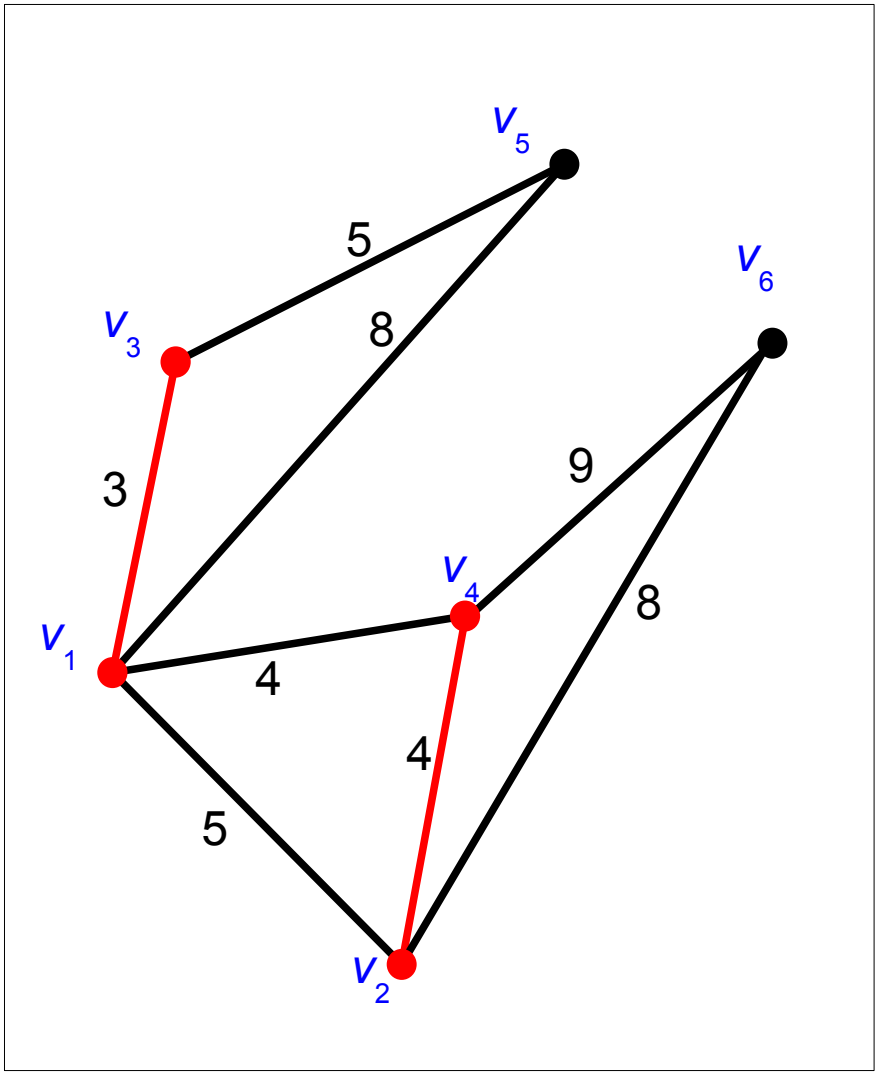


Algorytm KRUSKALA



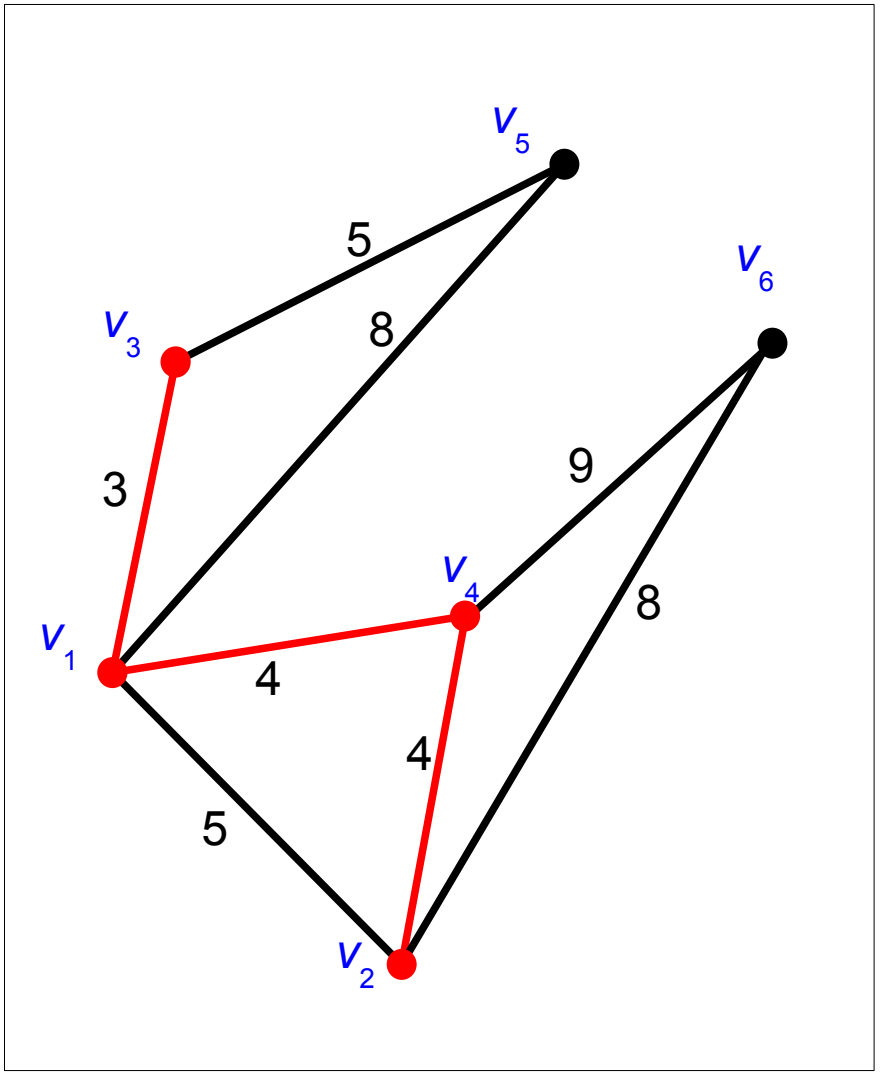
1. $(v_1, v_3) - 3$

Algorytm KRUSKALA



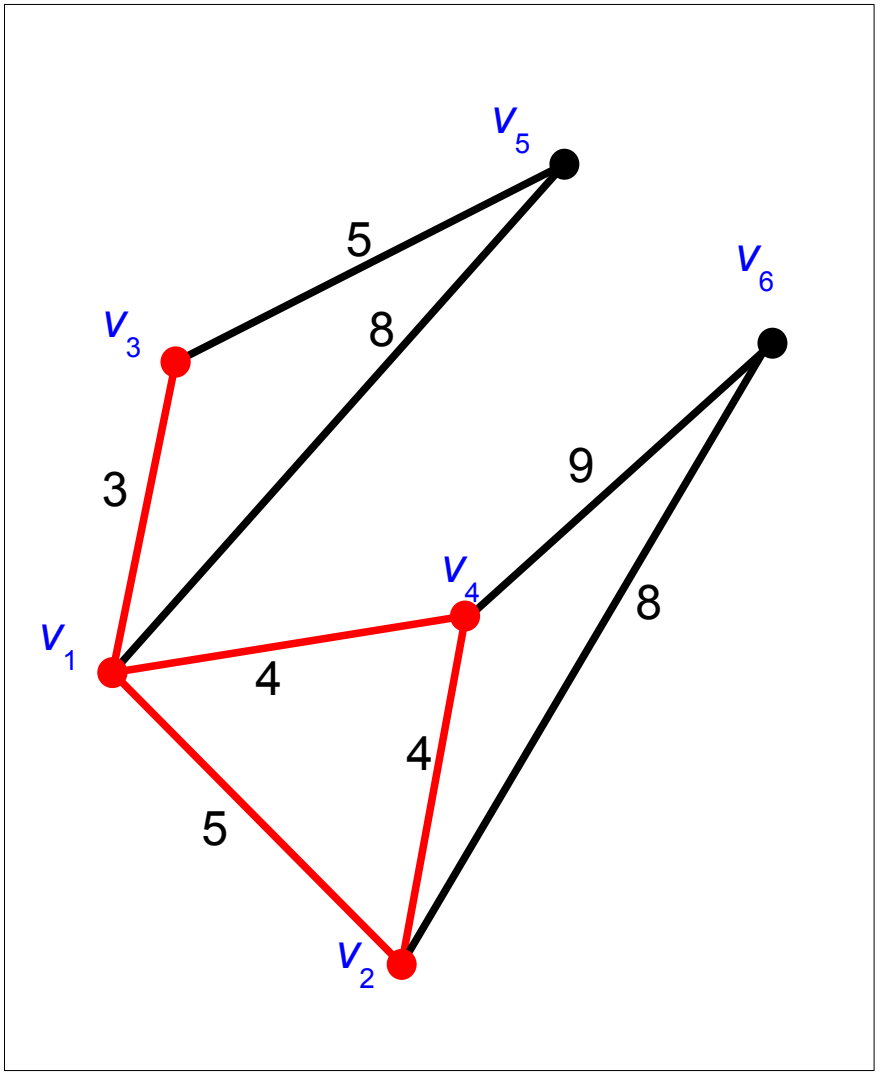
1. $(v_1, v_3) - 3$
2. $(v_2, v_4) - 4$

Algorytm KRUSKALA



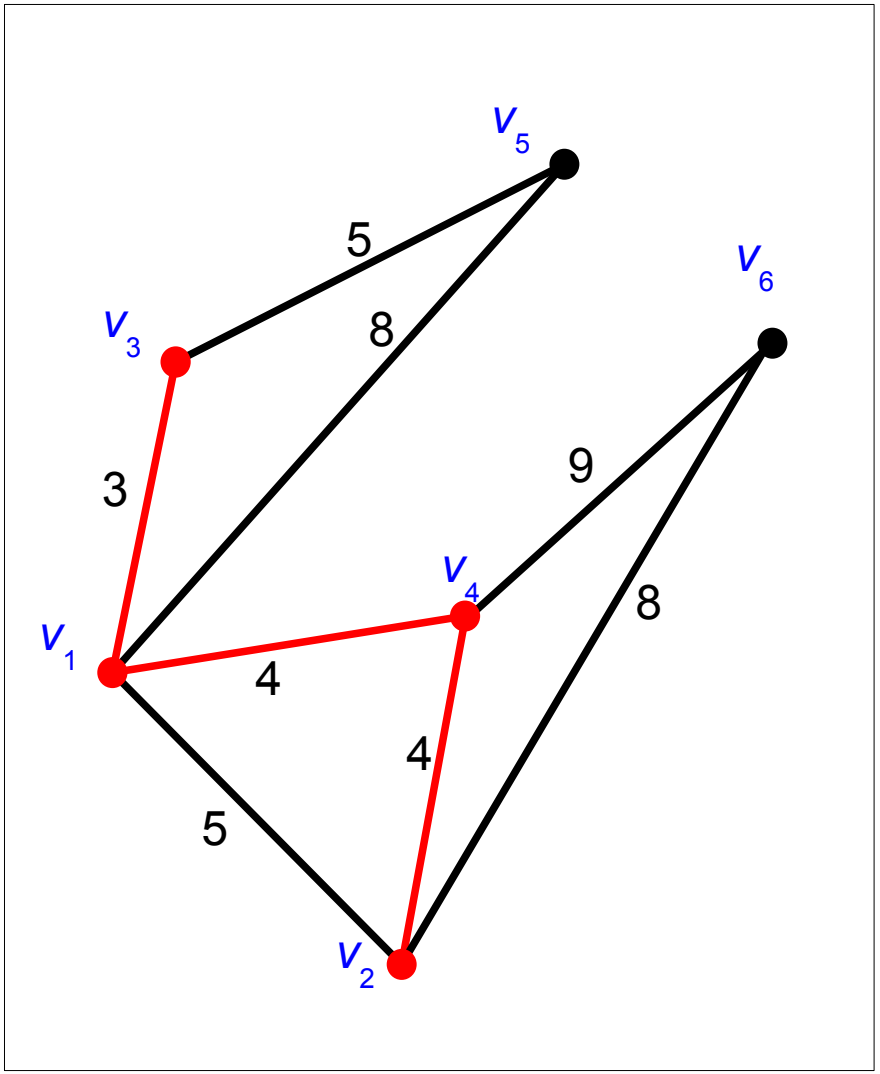
1. $(v_1, v_3) - 3$
2. $(v_2, v_4) - 4$
3. $(v_1, v_4) - 4$

Algorytm KRUSKALA



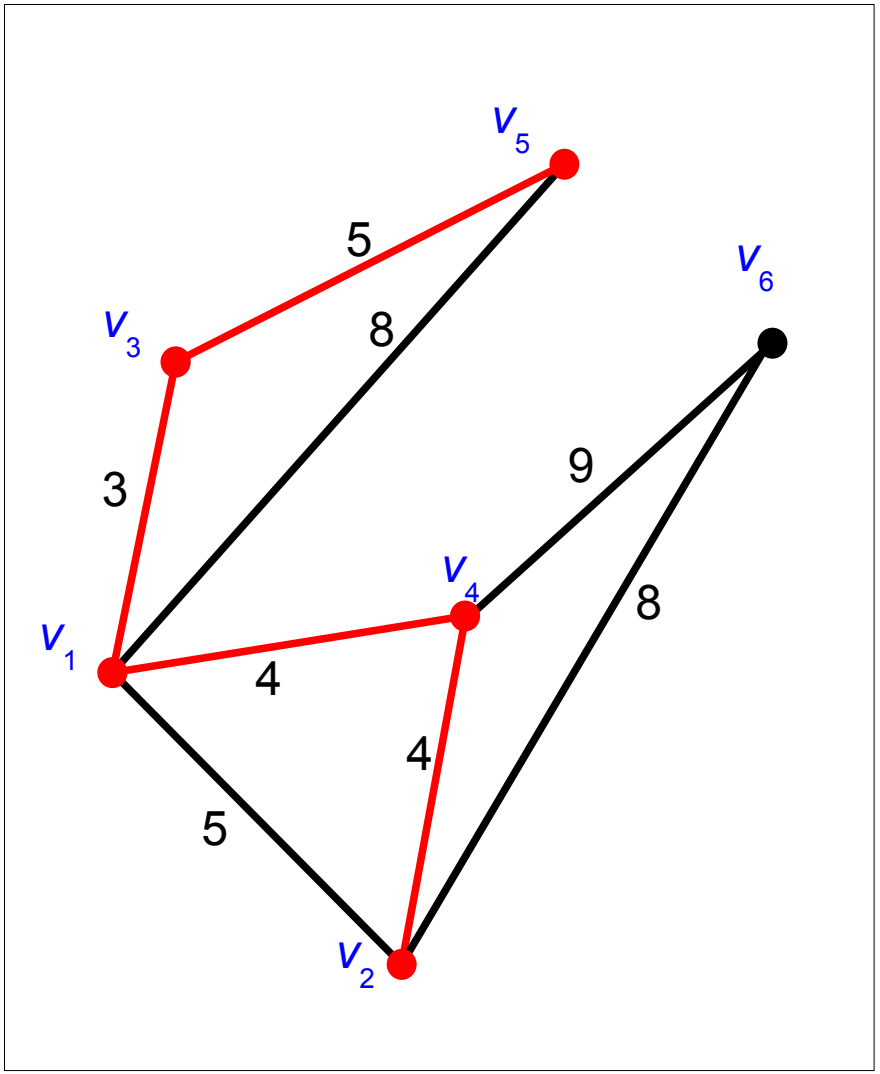
1. $(v_1, v_3) - 3$
2. $(v_2, v_4) - 4$
3. $(v_1, v_4) - 4$
4. (v_1, v_2) – cykl

Algorytm KRUSKALA



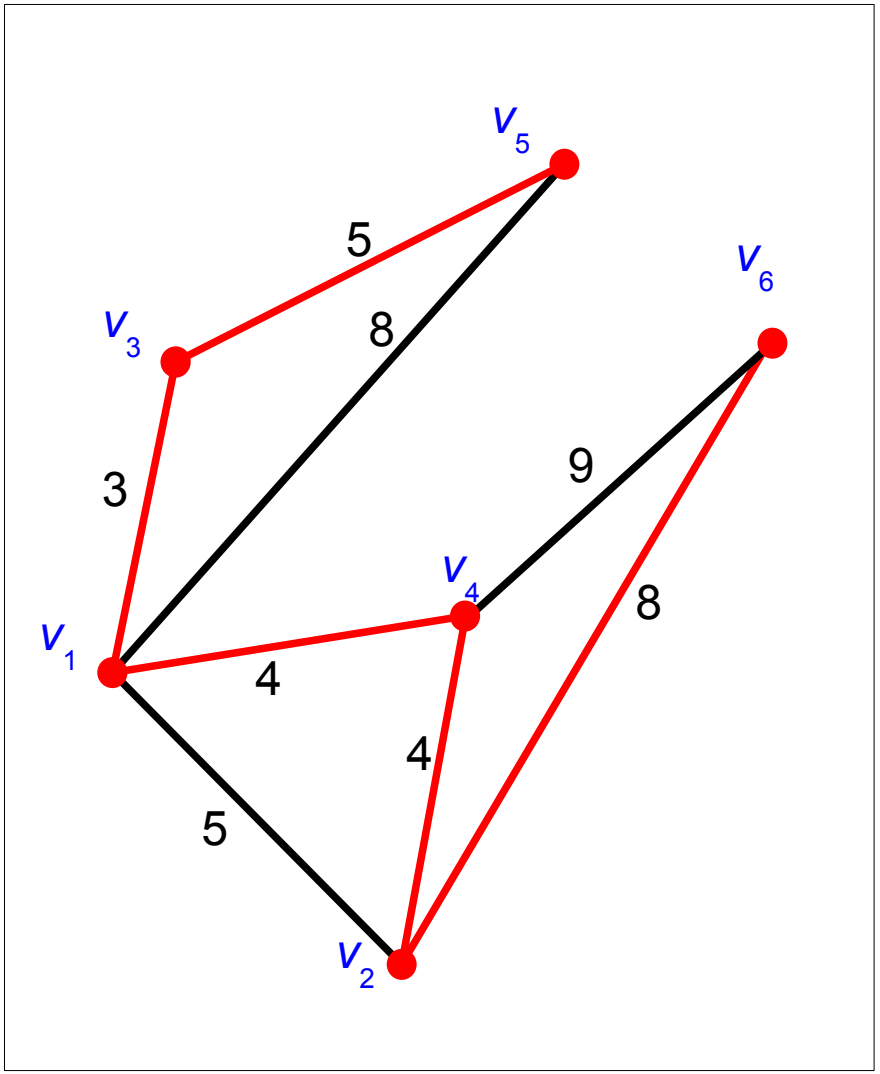
1. $(v_1, v_3) - 3$
2. $(v_2, v_4) - 4$
3. $(v_1, v_4) - 4$
4. (v_1, v_2) – cykl

Algorytm KRUSKALA



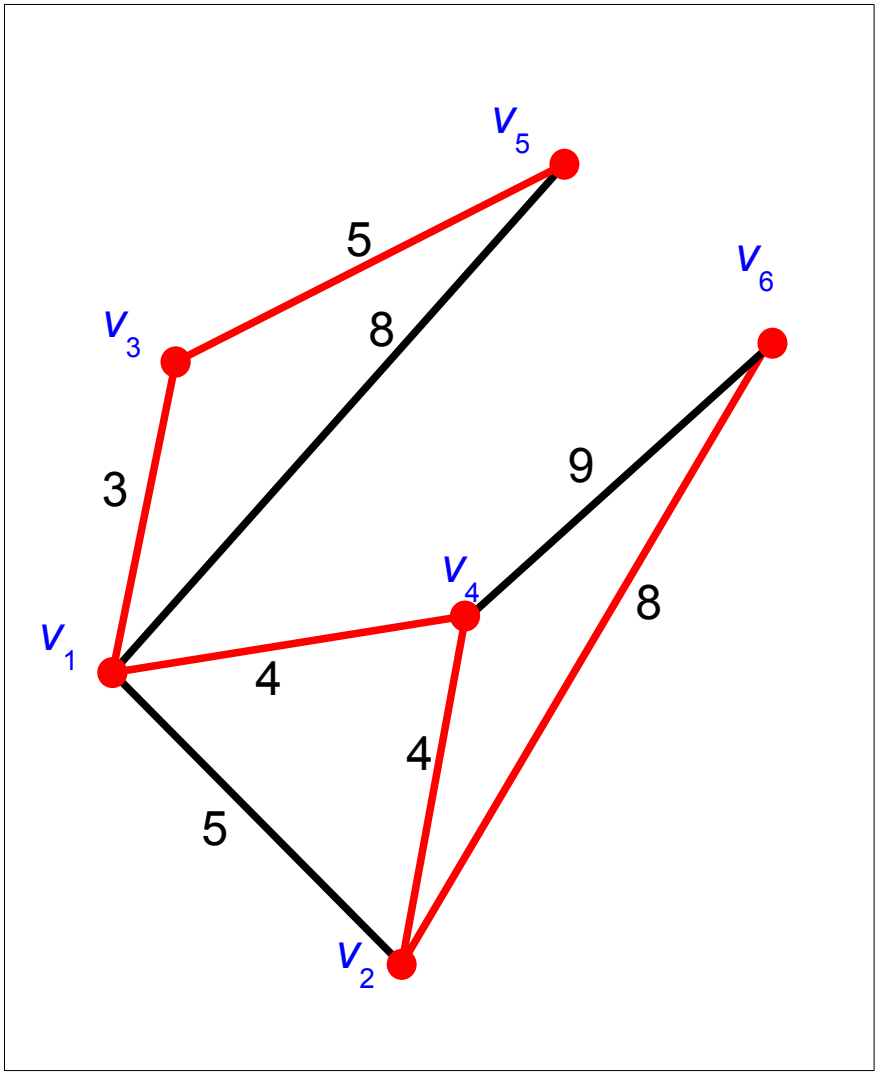
1. $(v_1, v_3) - 3$
2. $(v_2, v_4) - 4$
3. $(v_1, v_4) - 4$
4. (v_1, v_2) – cykl
5. $(v_3, v_5) - 5$

Algorytm KRUSKALA



1. $(v_1, v_3) - 3$
2. $(v_2, v_4) - 4$
3. $(v_1, v_4) - 4$
4. (v_1, v_2) – cykl
5. $(v_3, v_5) - 5$
6. $(v_2, v_6) - 8$

Algorytm KRUSKALA



1. $(v_1, v_3) - 3$
2. $(v_2, v_4) - 4$
3. $(v_1, v_4) - 4$
4. (v_1, v_2) – cykl
5. $(v_3, v_5) - 5$
6. $(v_2, v_6) - 8$

---

waga MST: 24

Algorytm PRIMA (1957)

rozpoczyna się od wybrania dowolnego wierzchołka i sukcesywnym dołączaniu najbliższego sąsiada (tj. wierzchołka, który jest połączony z którymś z już dołączonych krawędzią o najmniejszej wadze).

Dzięki temu, że w każdym kroku algorytmu dołączamy nowy wierzchołek do istniejącego poddrzewa, nigdy **nie spowoduje to powstania cyklu**, a wszystkich **iteracji** będzie $n - 1$.

Intuicyjnie, najkorzystniej jest zastosować do tego algorytmu struktury pozwalające na efektywne wyszukiwanie wierzchołków incydentnych w danym, a więc np. **połączone listy sąsiadów**.

Znalezienie kolejnego wierzchołka do dołączenia będzie w takim wypadku wymagało przejrzenia list struktury odpowiadających wierzchołkom już dołączonym i wybraniu krawędzi o najmniejszej wadze. W najgorszym wypadku będzie więc musieli przejrzeć wszystkie krawędzie, a więc **pojedyncza iteracja** zajmie $O(m)$ czasu.

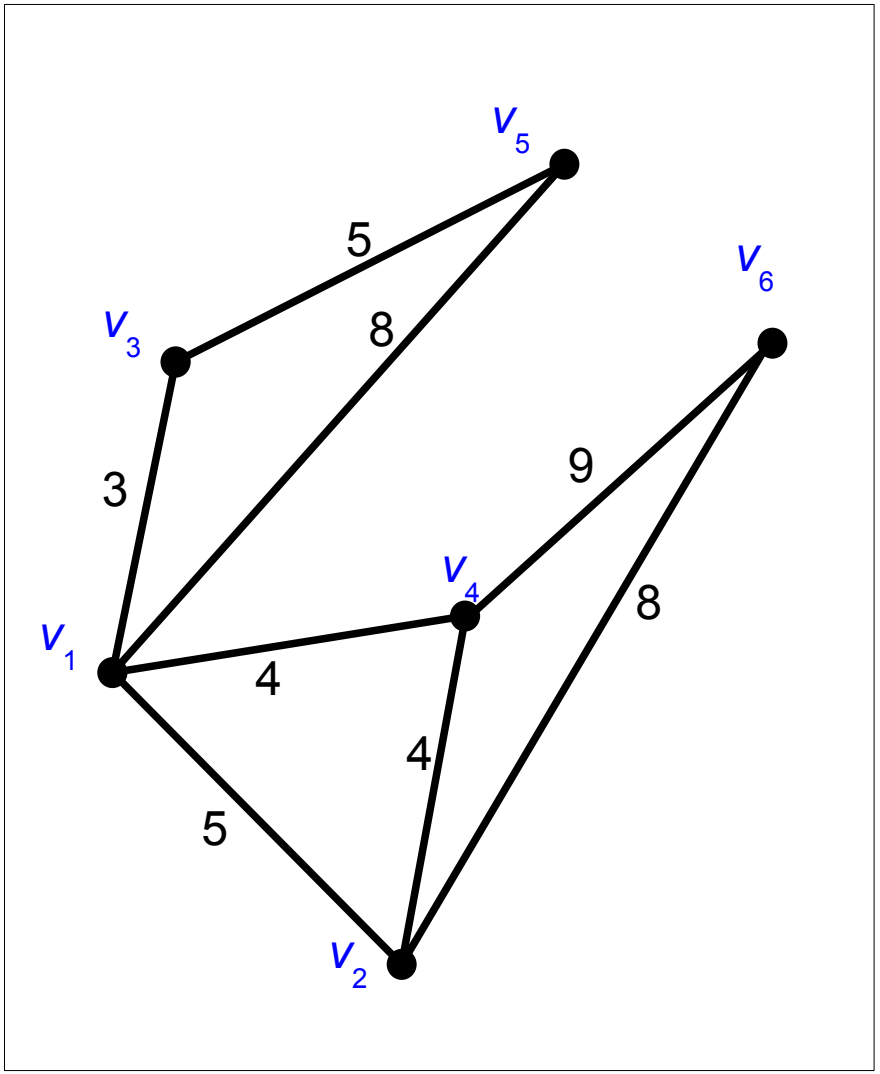
Zatem **złożoność** całego **algorytmu Prima** to $O(nm)$.

Algorytm Prima można też zaimplementować w taki sposób, że **wierzchołki oczekujące** na dołączenie umieścimy w **kolejce Q**:
początkowo zawiera ona wszystkie wierzchołki, w każdej iteracji pobierany jest dokładnie jeden, a algorytm kończy działanie, gdy stanie się ona pusta.

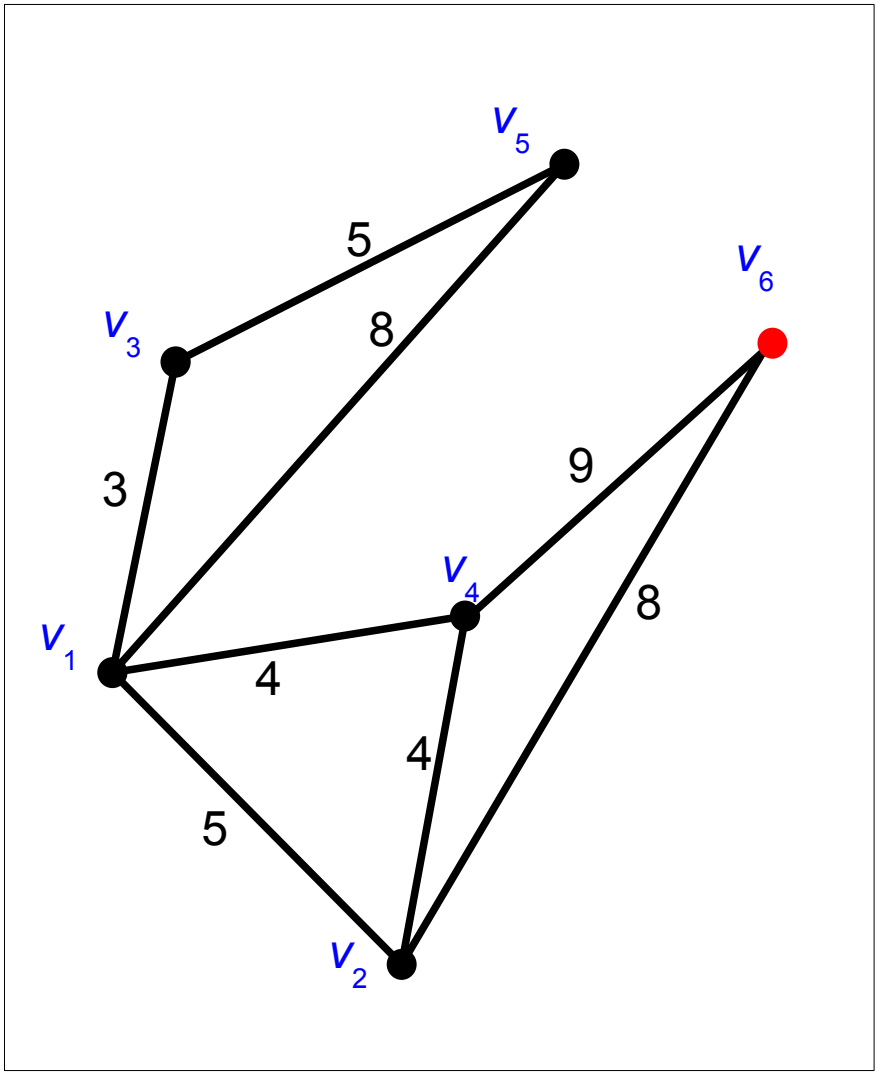
Kolejkę tą można zaimplementować jako kopiec, w którym wartością węzła jest odległość danego wierzchołka od najbliższego wierzchołka spośród już dołączonych. Wartości te należy aktualizować po każdym dołączeniu nowego wierzchołka, ale za to mamy od razu dostępną informację, który wierzchołek ma być dołączony jako następny. Takie podejście umożliwia uzyskanie złożoności obliczeniowej algorytmu Prima $O(m \log n)$.

Z kolei, jeśli zamiast zwykłego kopca, użyjemy kopca Fibonacciego, uzyskamy złożoność $O(m + n \log n)$.

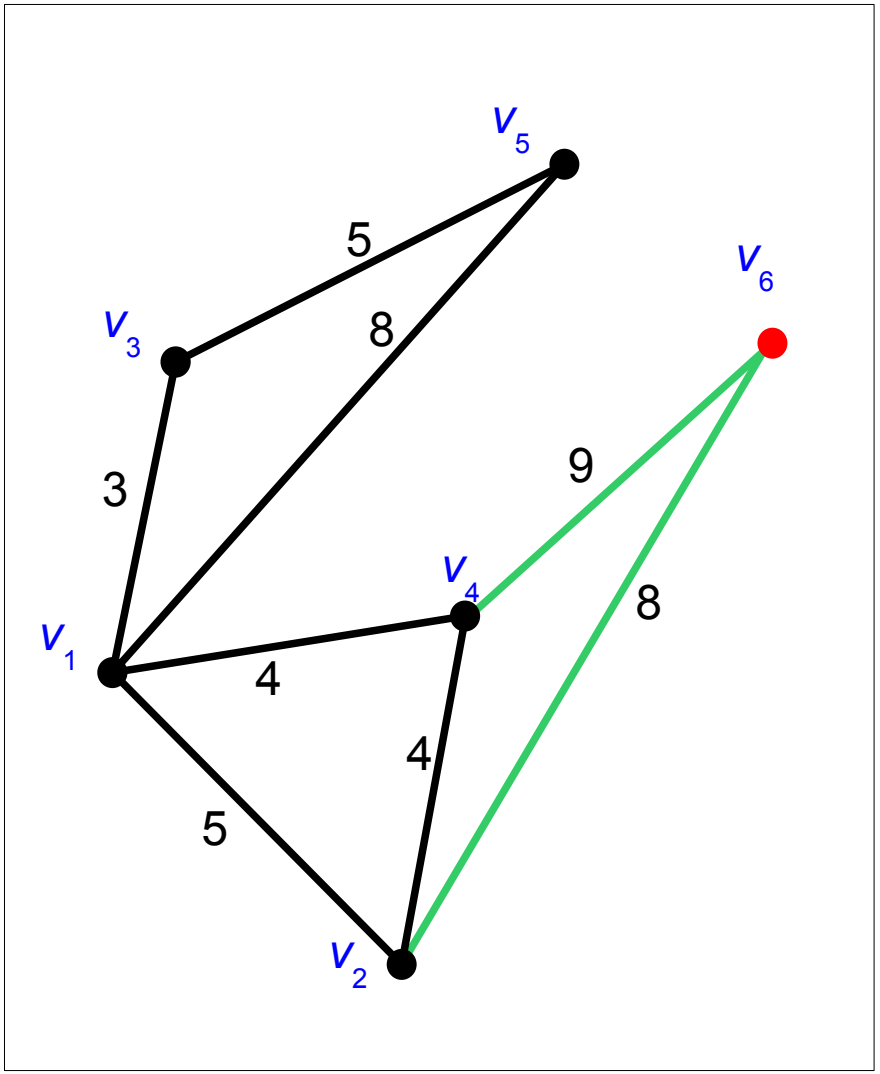
Algorytm PRIMA



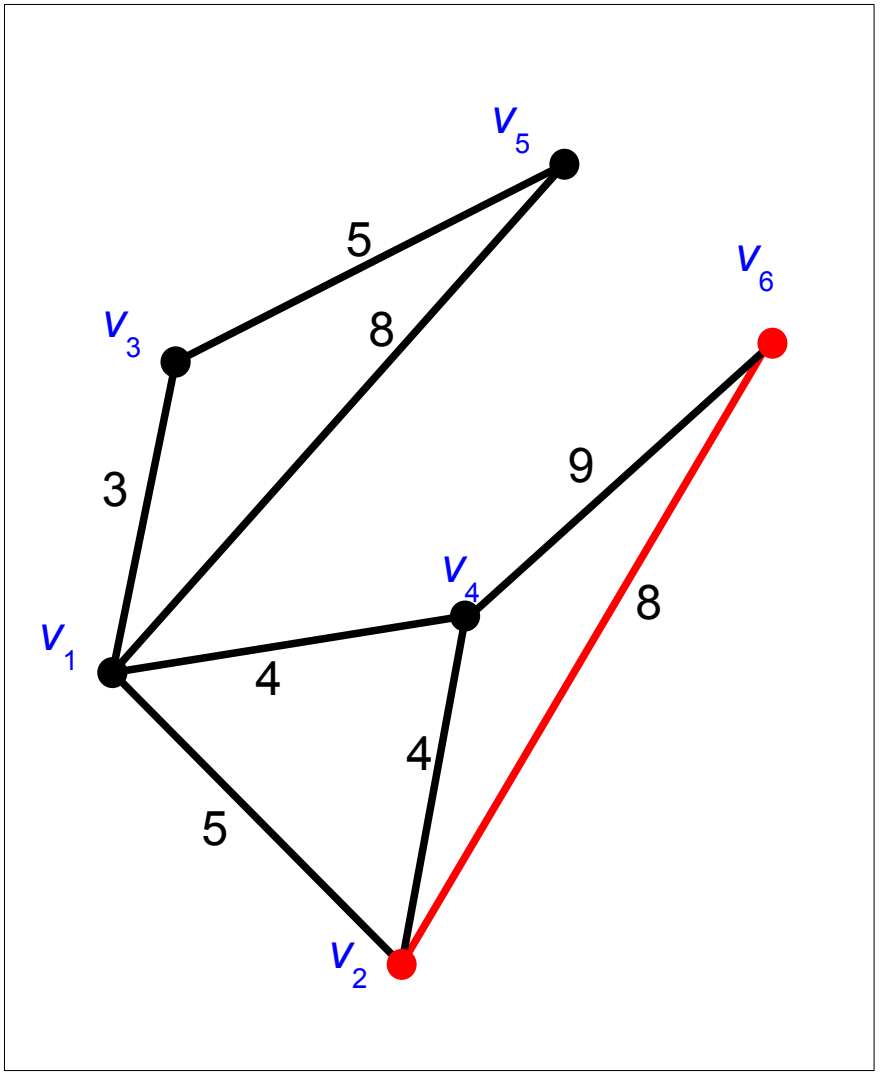
Algorytm PRIMA



Algorytm PRIMA

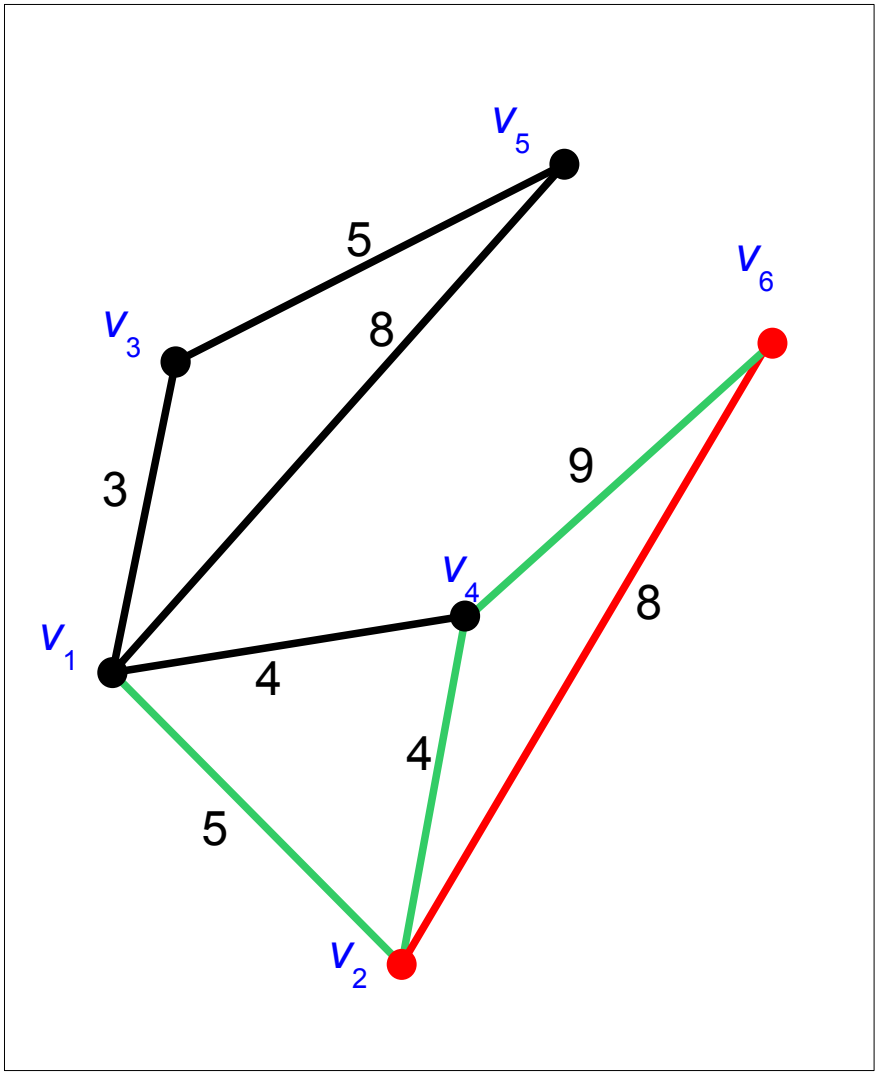


Algorytm PRIMA



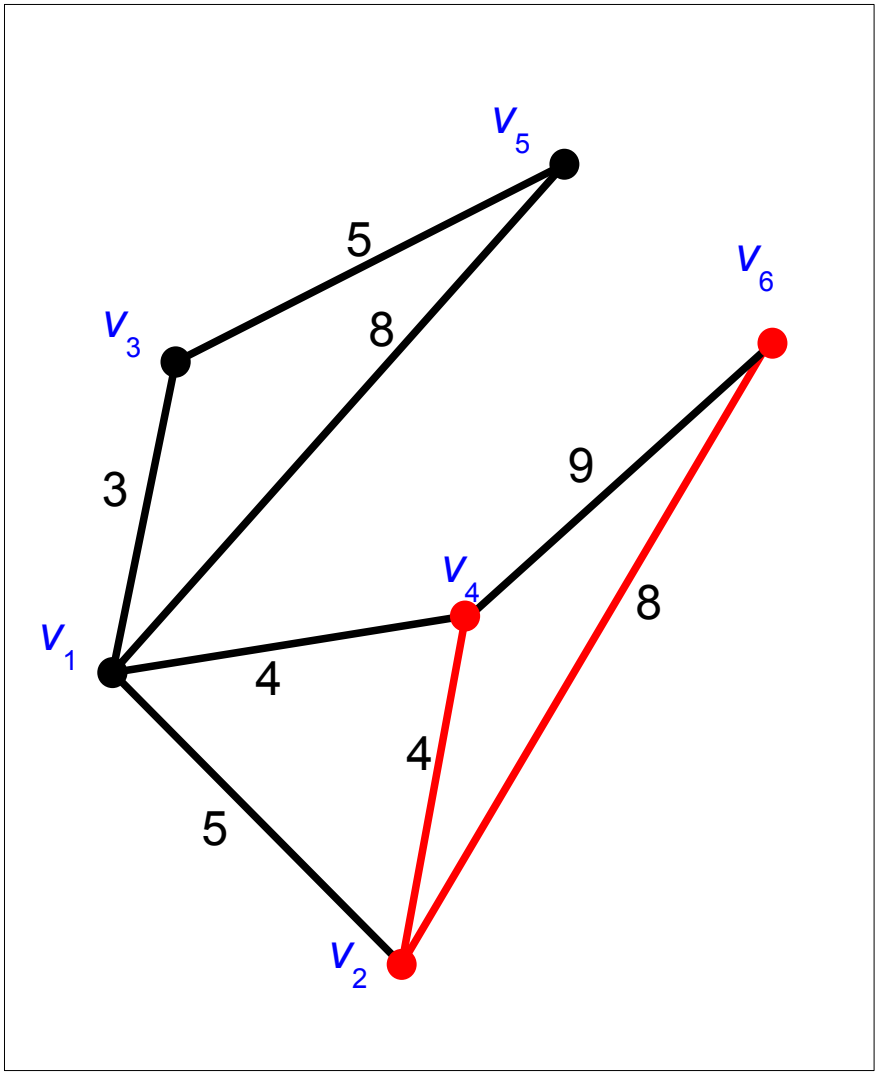
1. $(v_6, v_2) - 8$

Algorytm PRIMA



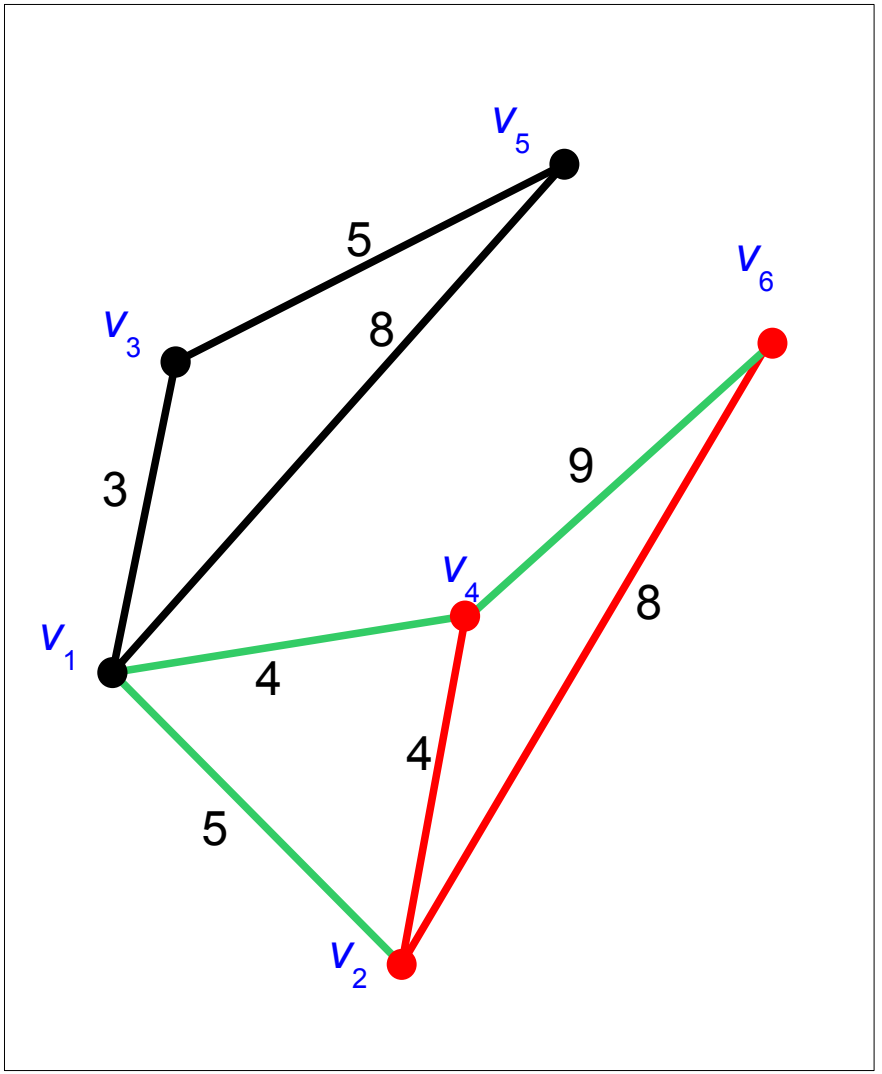
1. $(v_6, v_2) - 8$

Algorytm PRIMA



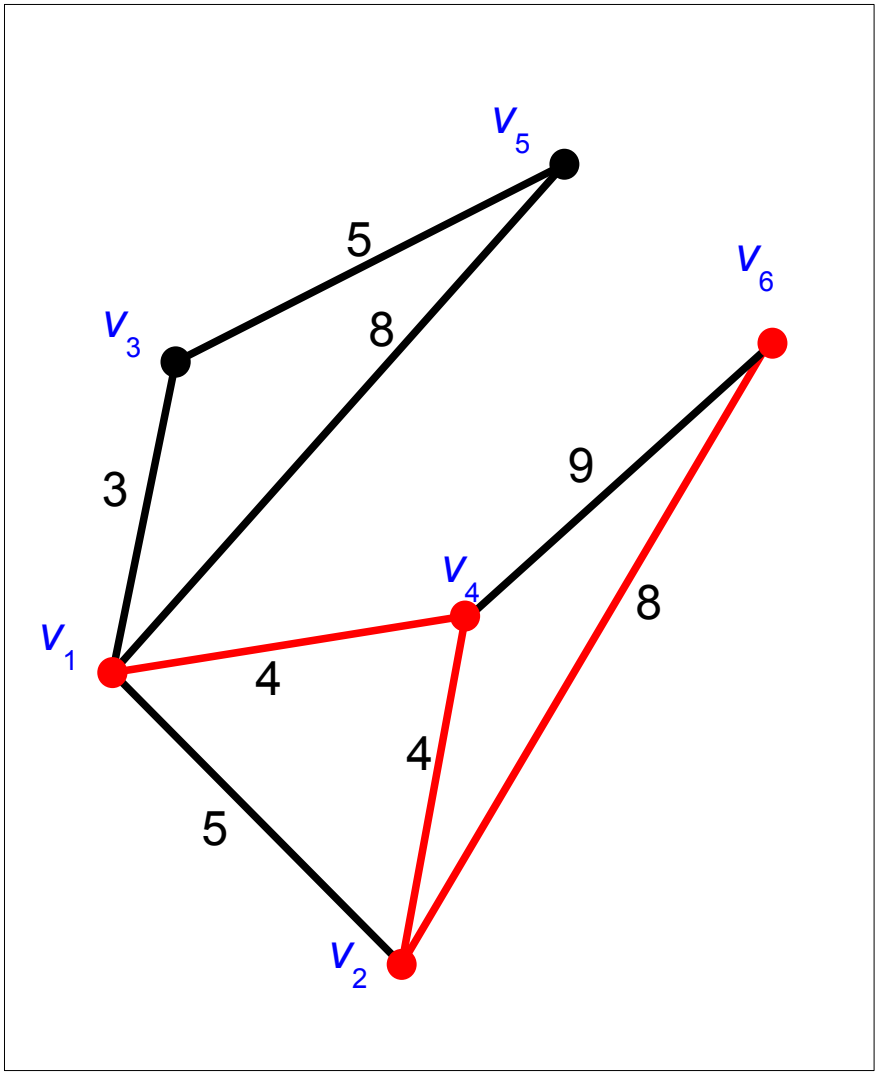
1. $(v_6, v_2) - 8$
2. $(v_2, v_4) - 4$

Algorytm PRIMA



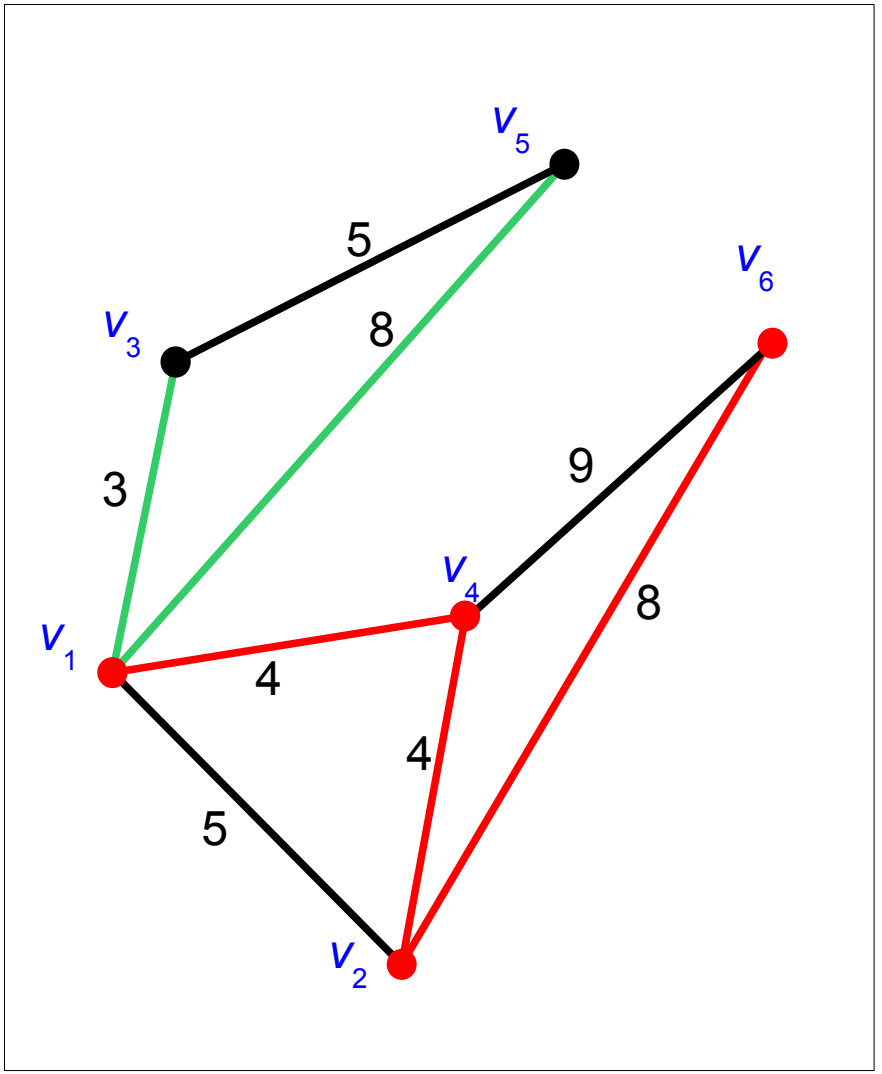
1. $(v_6, v_2) - 8$
2. $(v_2, v_4) - 4$

Algorytm PRIMA



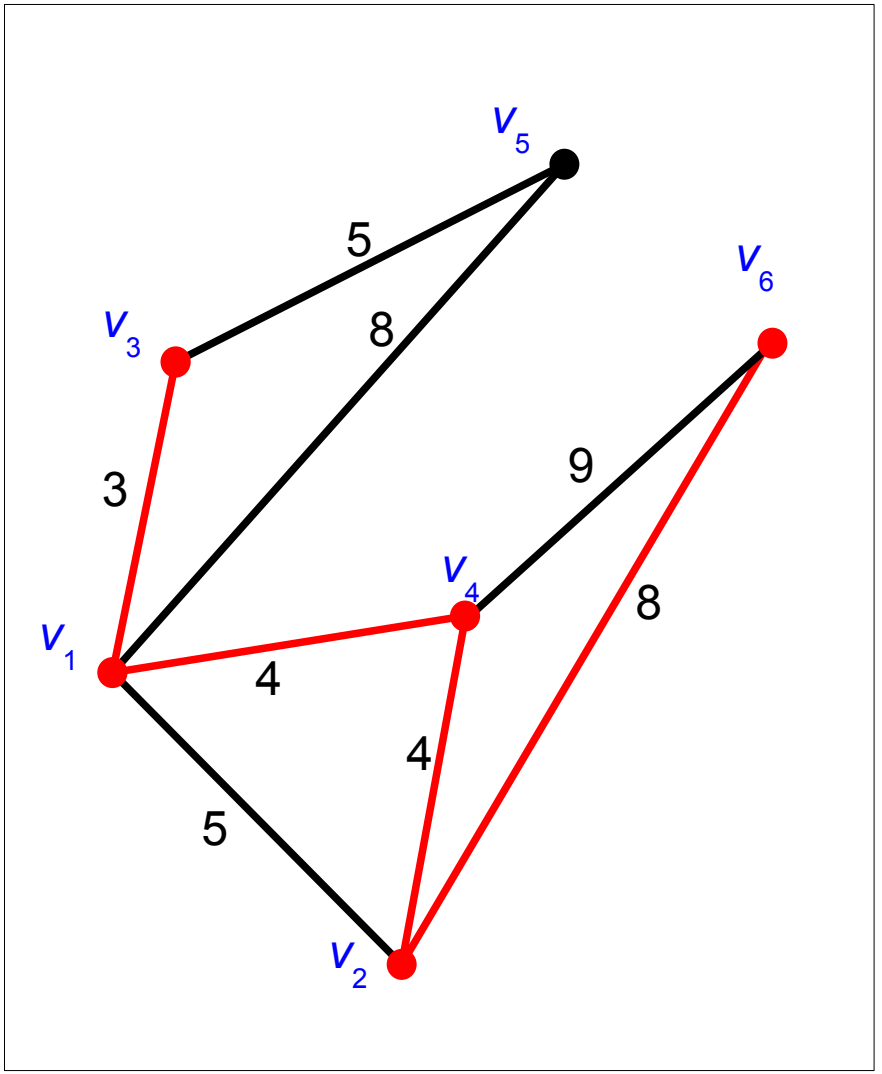
1. $(v_6, v_2) - 8$
2. $(v_2, v_4) - 4$
3. $(v_1, v_4) - 4$

Algorytm PRIMA



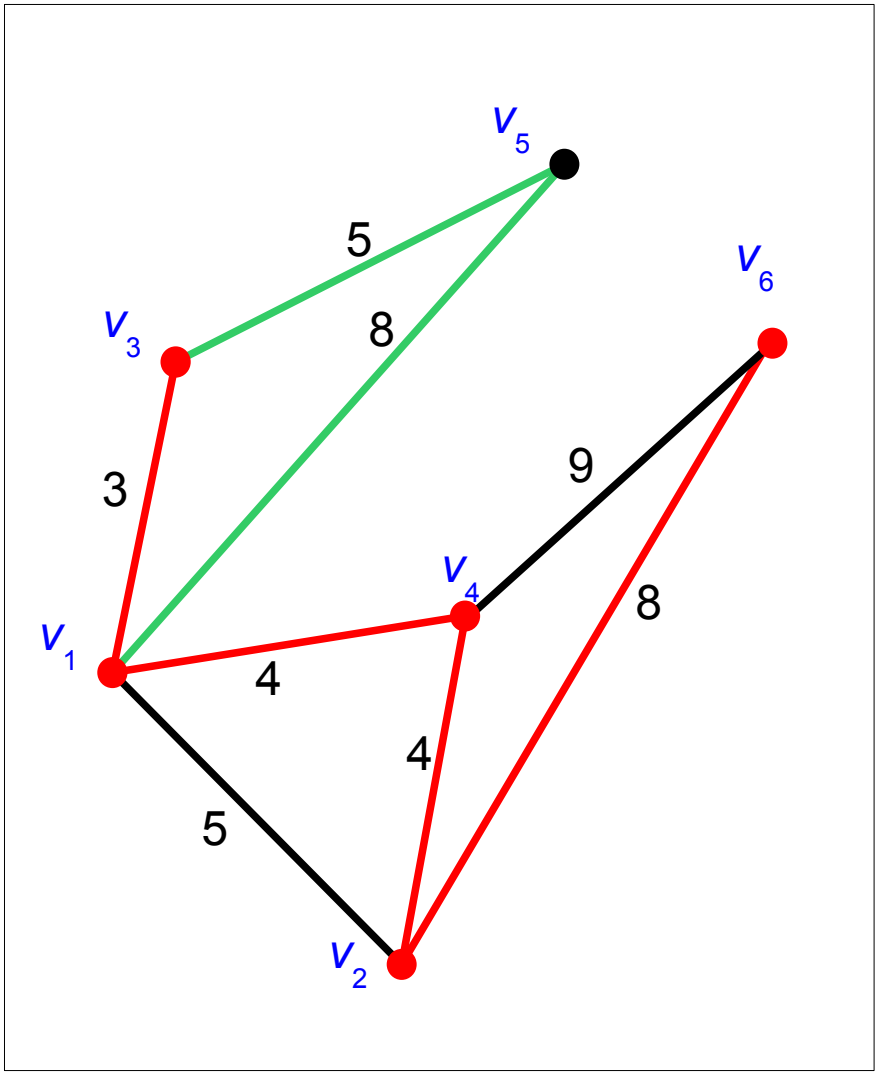
1. $(v_6, v_2) - 8$
2. $(v_2, v_4) - 4$
3. $(v_1, v_4) - 4$

Algorytm PRIMA



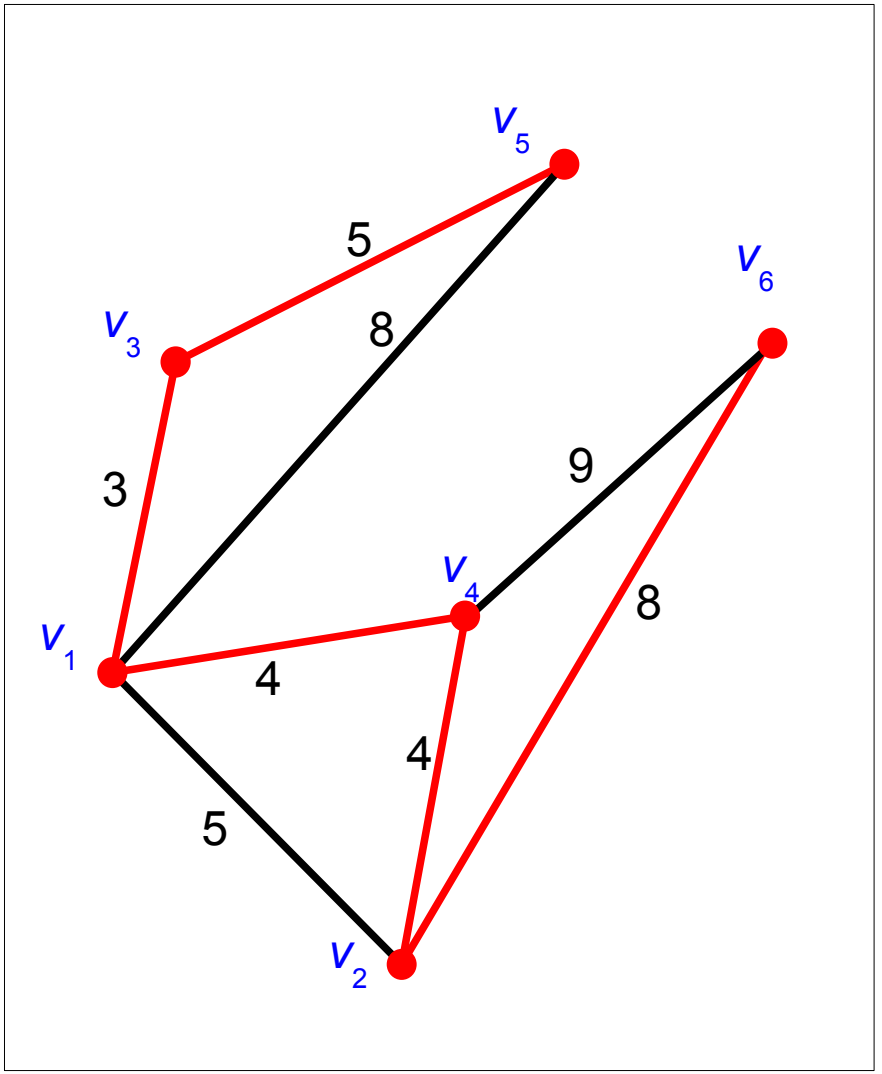
1. $(v_6, v_2) - 8$
2. $(v_2, v_4) - 4$
3. $(v_1, v_4) - 4$
4. $(v_1, v_3) - 3$

Algorytm PRIMA



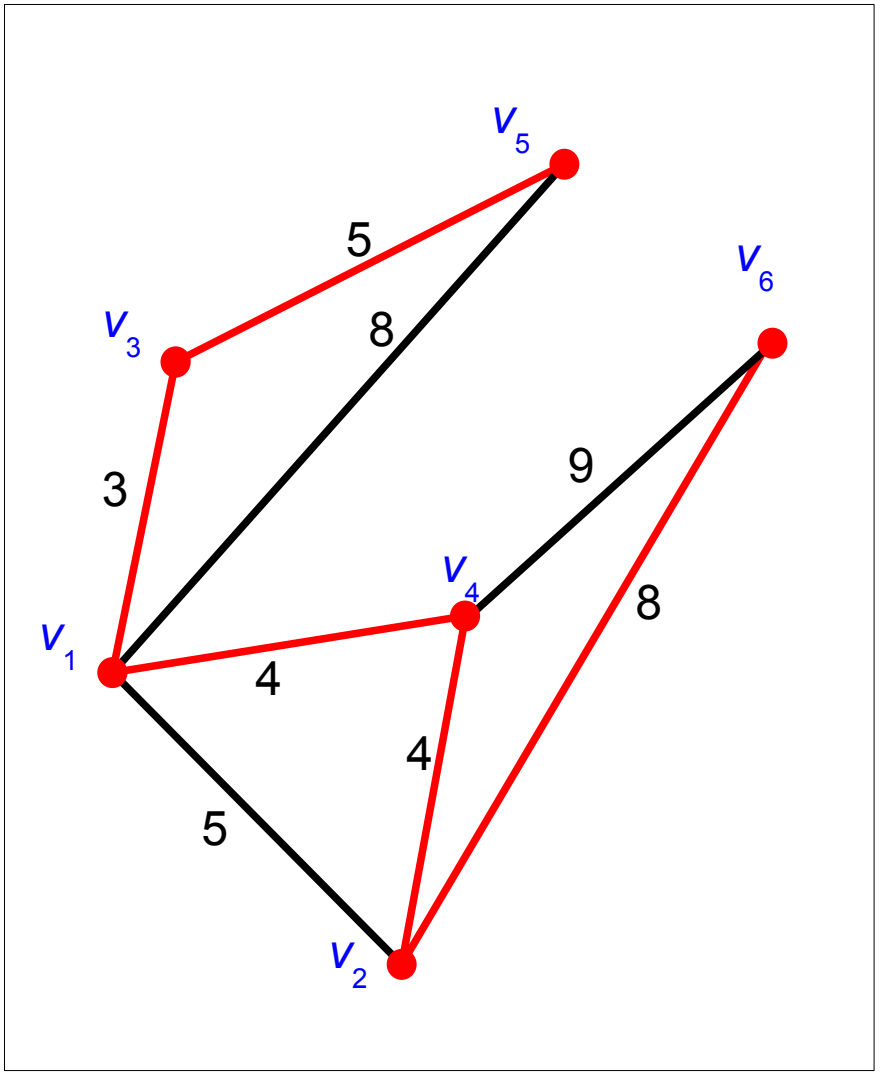
1. $(v_6, v_2) - 8$
2. $(v_2, v_4) - 4$
3. $(v_1, v_4) - 4$
4. $(v_1, v_3) - 3$

Algorytm PRIMA



1. $(v_6, v_2) - 8$
2. $(v_2, v_4) - 4$
3. $(v_1, v_4) - 4$
4. $(v_1, v_3) - 3$
5. $(v_3, v_5) - 5$

Algorytm PRIMA



1. $(v_6, v_2) - 8$
2. $(v_2, v_4) - 4$
3. $(v_1, v_4) - 4$
4. $(v_1, v_3) - 3$
5. $(v_3, v_5) - 5$

---

waga MST: 24

Algorytmy wyznaczania najkrótszych dróg

Algorytm DIJKSTRY (1959)

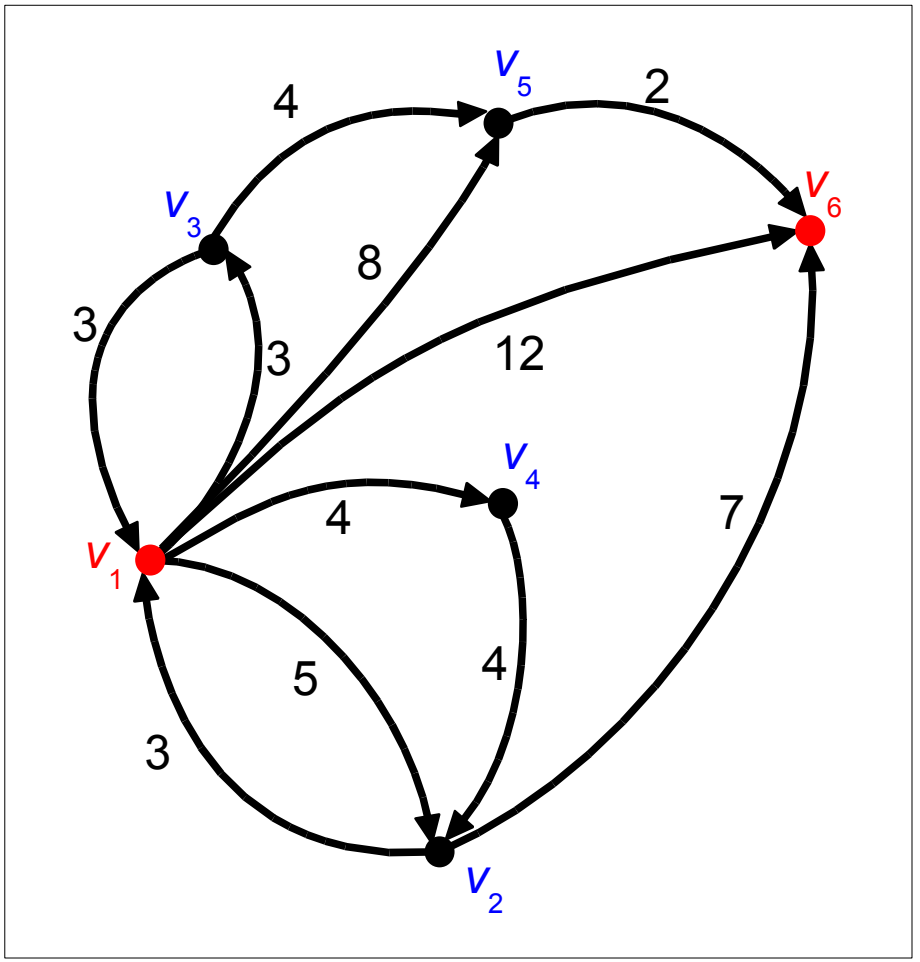
służy do wyznaczania najkrótszych dróg z wyznaczonego węzła (źródła, s) do wszystkich pozostałych węzłów, w przypadku gdy **wagi** wszystkich łuków grafu są **nieujemne**.

W trakcie działania algorytmu, dla każdego węzła, x, pamiętane jest ***oszacowanie* wagi najkrótszej drogi** ze źródła: $\boldsymbol{d_{sx}}$. Przed rozpoczęciem działania algorytmu wartości te wynoszą ∞ dla wszystkich węzłów z wyjątkiem źródła, dla którego $d_{ss} = 0$.

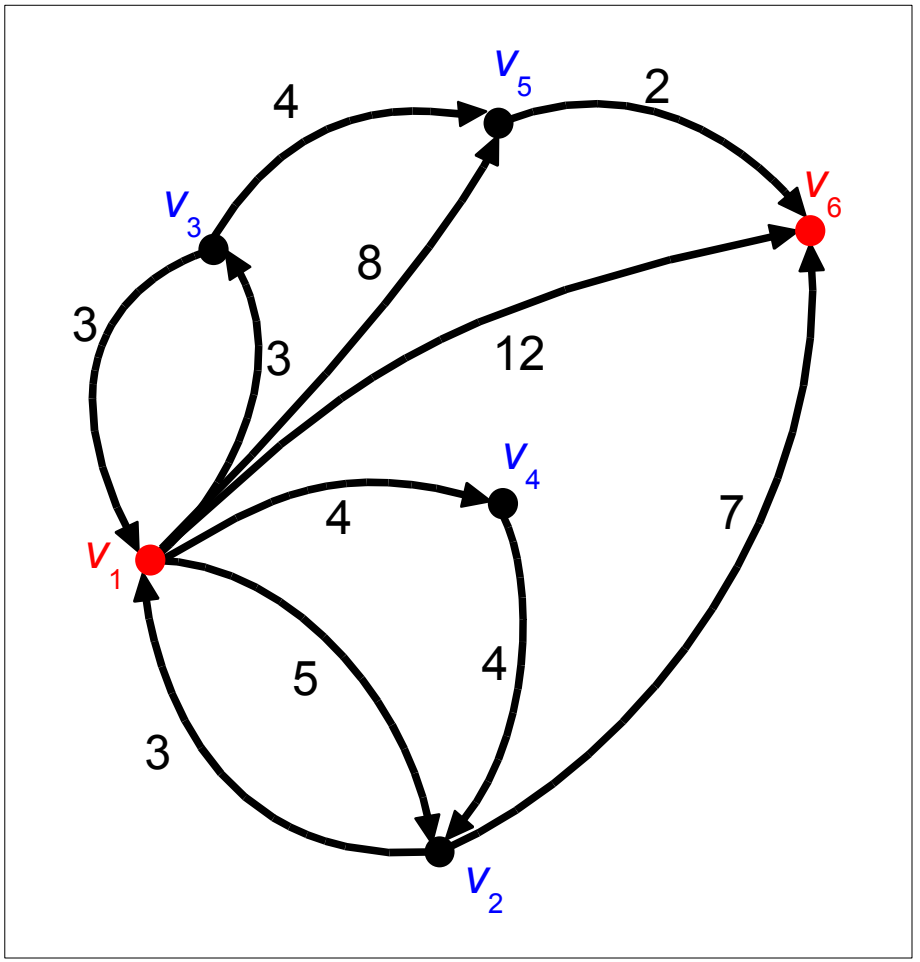
W każdym kroku znajdowany jest węzeł x^*, dla którego **oszacowanie** to jest **minimalne** i nieustalone i jest ono *ustalane* (oznacza to, że dla tego węzła znaleźliśmy już drogę minimalną).

Następnie dokonuje się **relaksacji** wszystkich **łuków wychodzących** z węzła x^*, tzn. jeśli aktualne oszacowanie, d_{sy}, badanego węzła y, jest większe niż suma oszacowania d_{sx^*} oraz wagi, w_{x^*y}, łuku $\{x^*, y\}$ to $d_{sy} = d_{sx^*} + w_{x^*y}$.

Algorytm DIJKSTRY

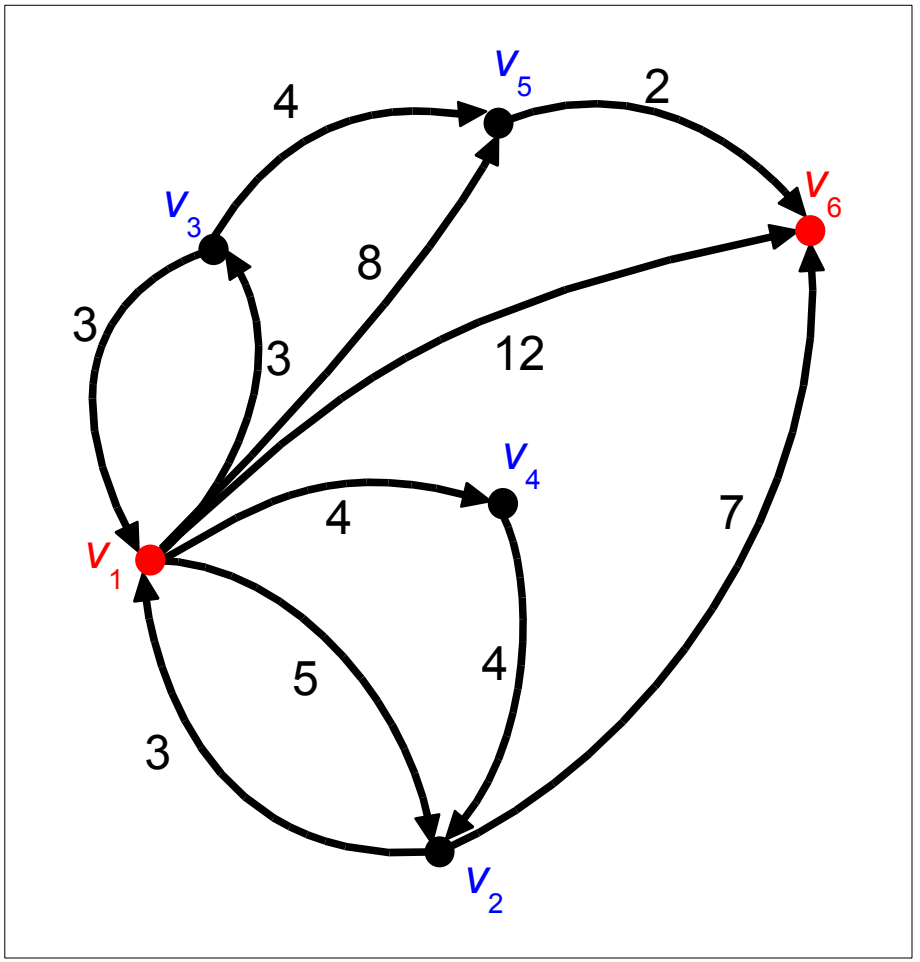


Algorytm DIJKSTRY



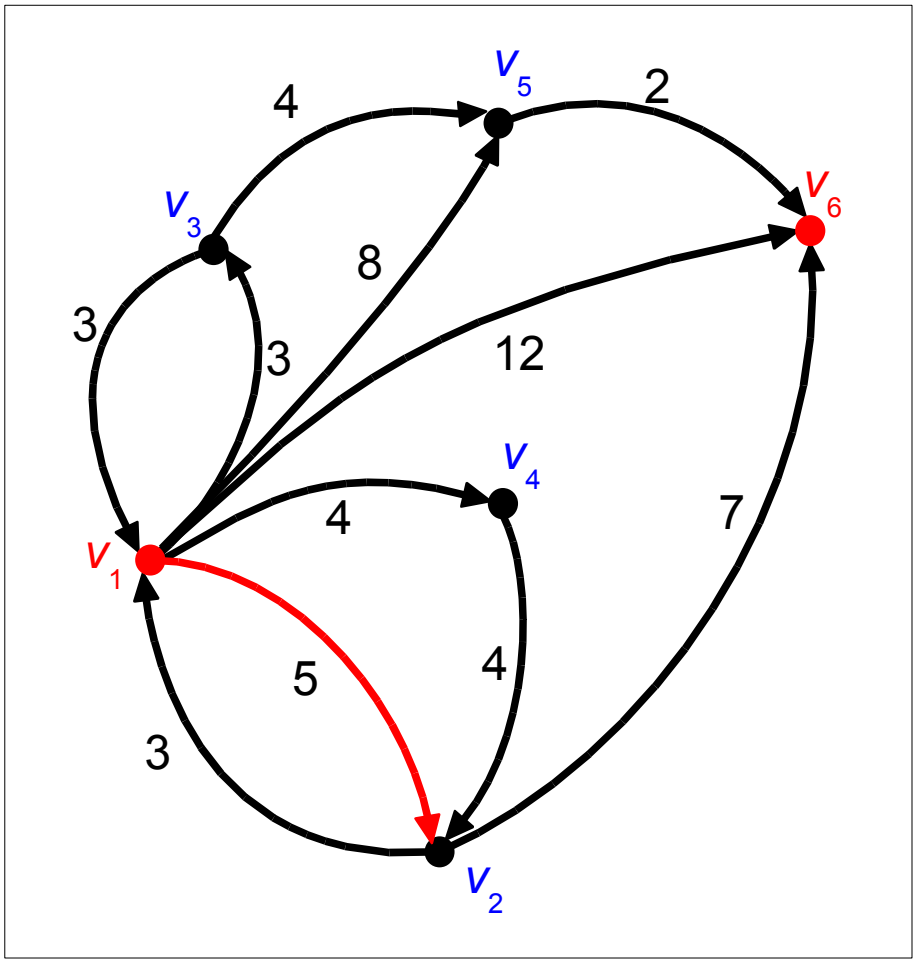
Krok	v_1	v_2	v_3	v_4	v_5	v_6
1	0	∞	∞	∞	∞	∞

Algorytm DIJKSTRY



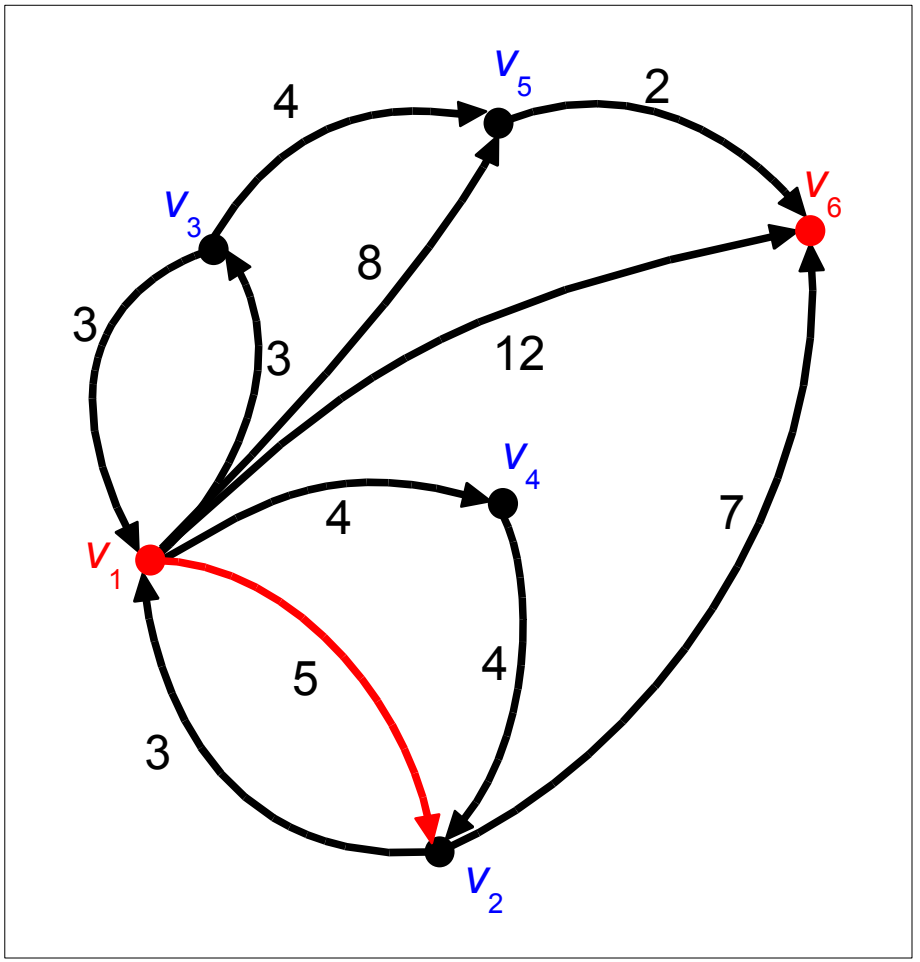
Krok	v_1	v_2	v_3	v_4	v_5	v_6
1	<u>0</u>	∞	∞	∞	∞	∞

Algorytm DIJKSTRY



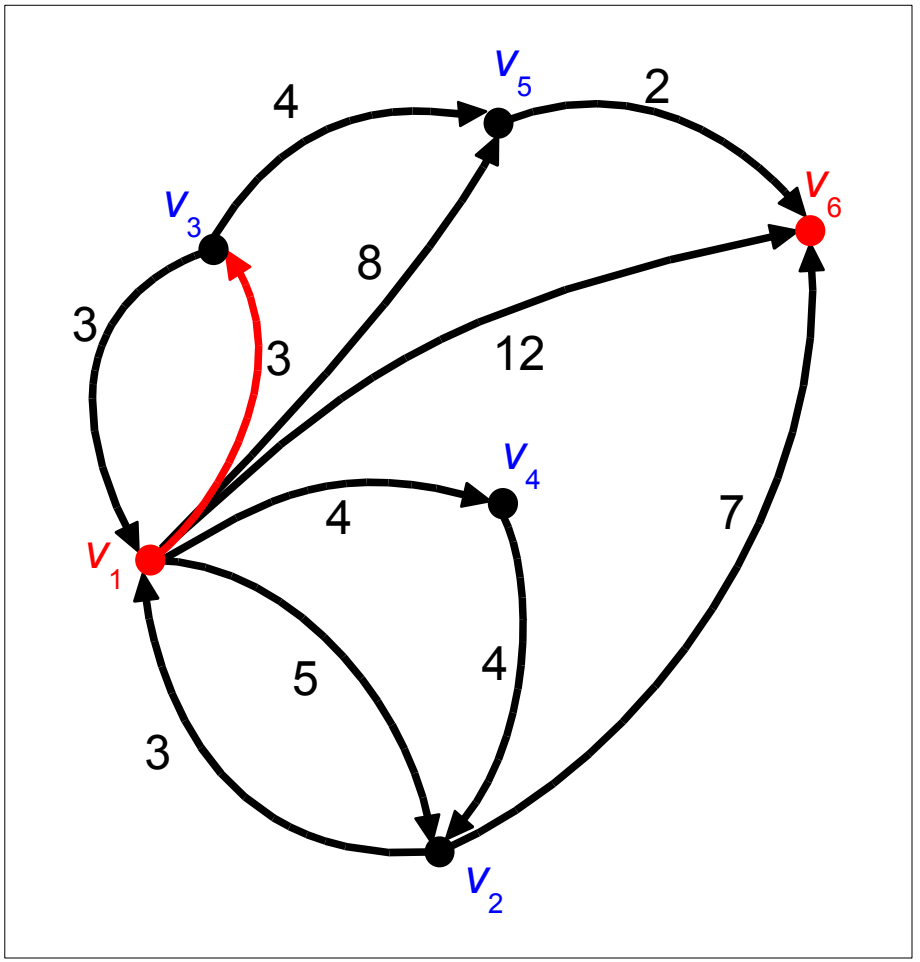
Krok	v_1	v_2	v_3	v_4	v_5	v_6
1	0	∞	∞	∞	∞	∞

Algorytm DIJKSTRY



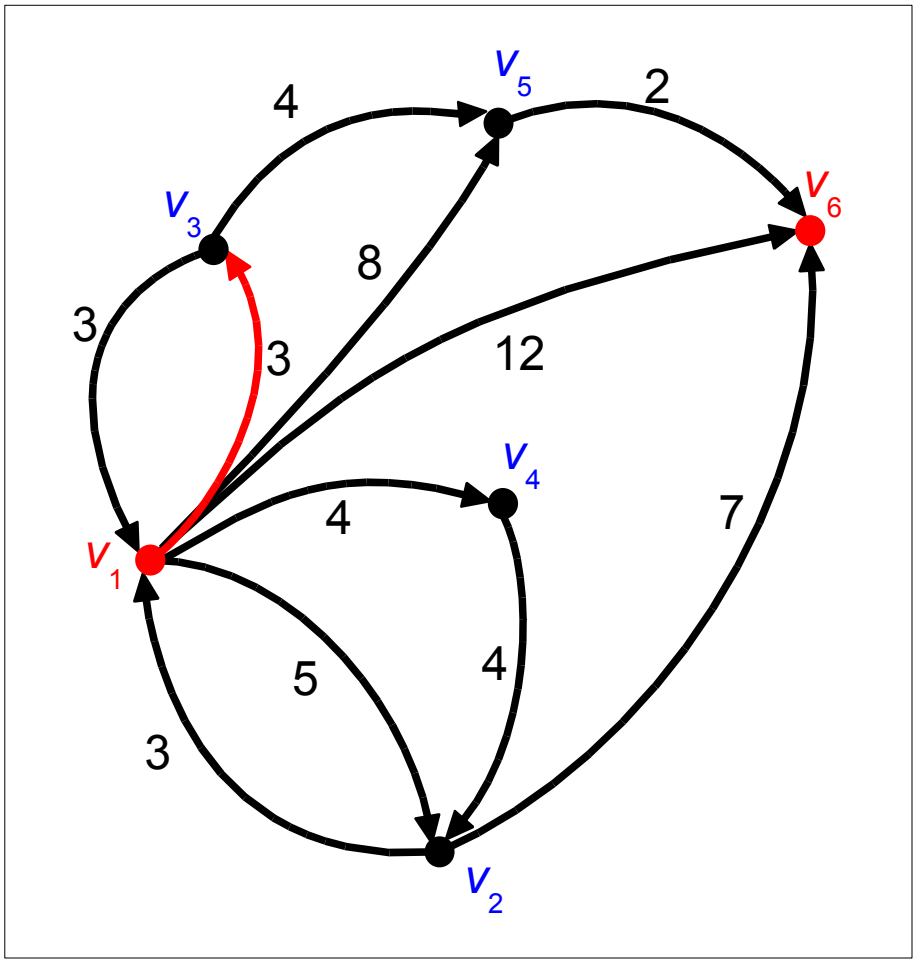
Krok	v_1	v_2	v_3	v_4	v_5	v_6
1	0	5	∞	∞	∞	∞

Algorytm DIJKSTRY



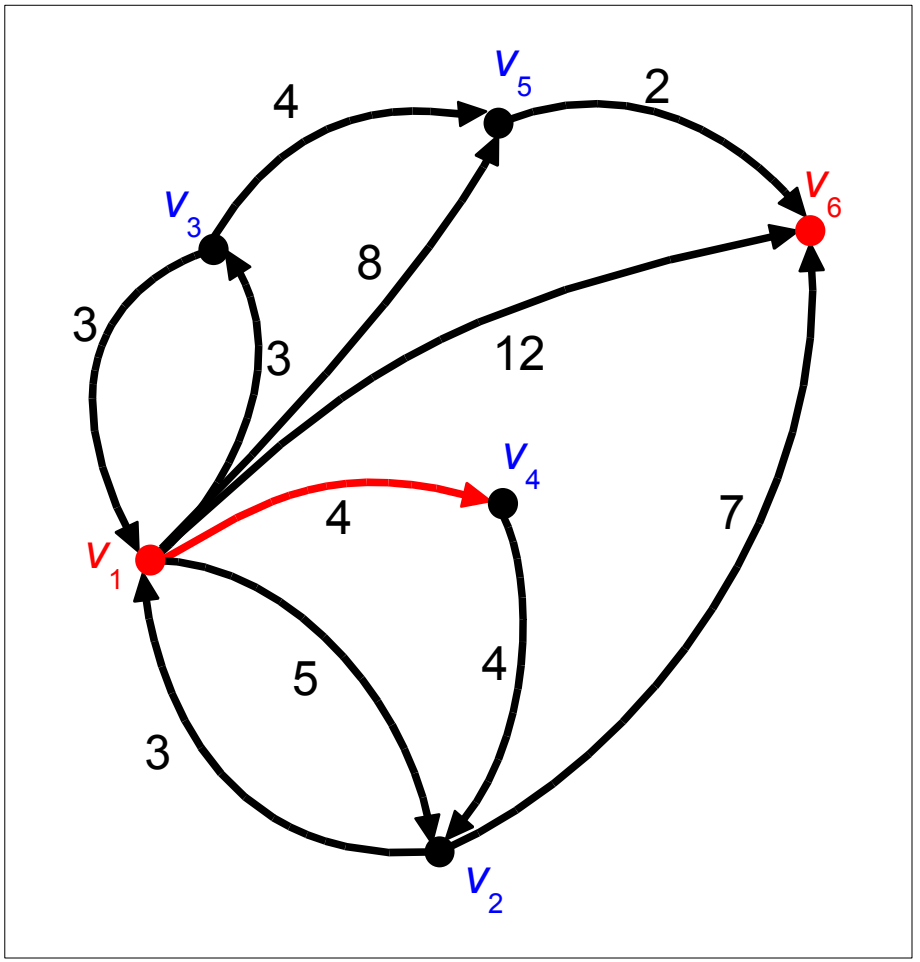
Krok	v_1	v_2	v_3	v_4	v_5	v_6
1	<u>0</u>	5	∞	∞	∞	∞

Algorytm DIJKSTRY



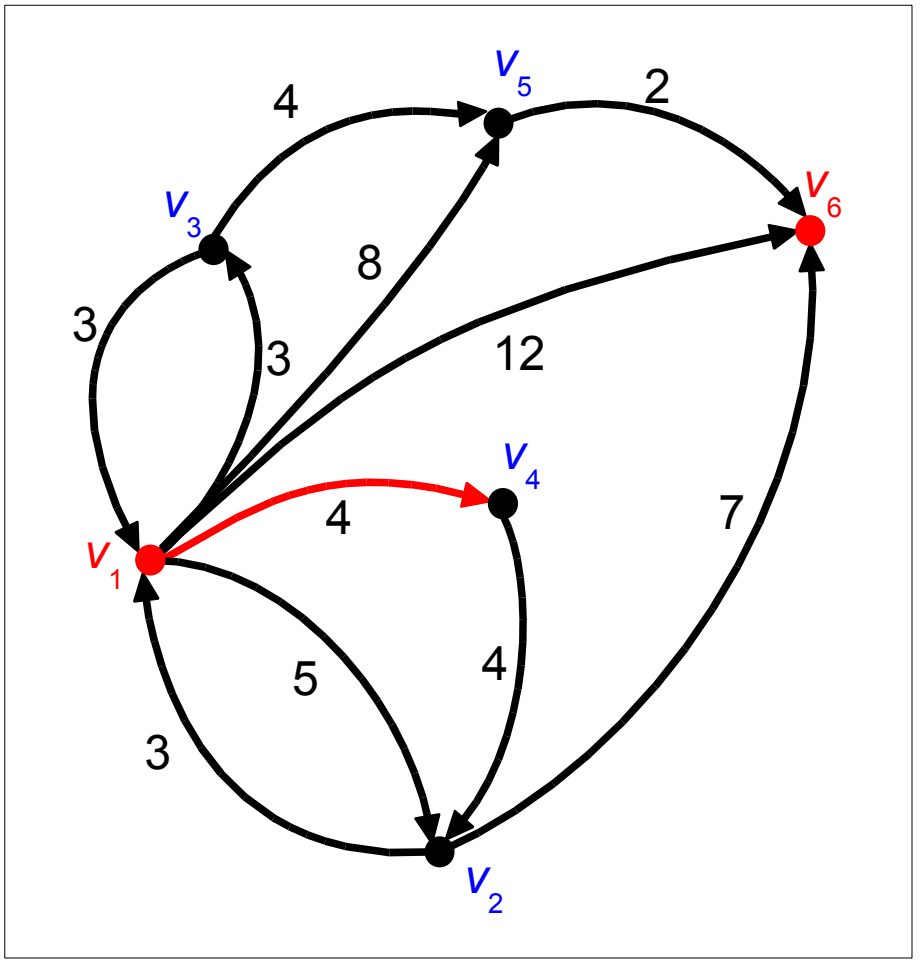
Krok	v_1	v_2	v_3	v_4	v_5	v_6
1	<u>0</u>	5	3	∞	∞	∞

Algorytm DIJKSTRY



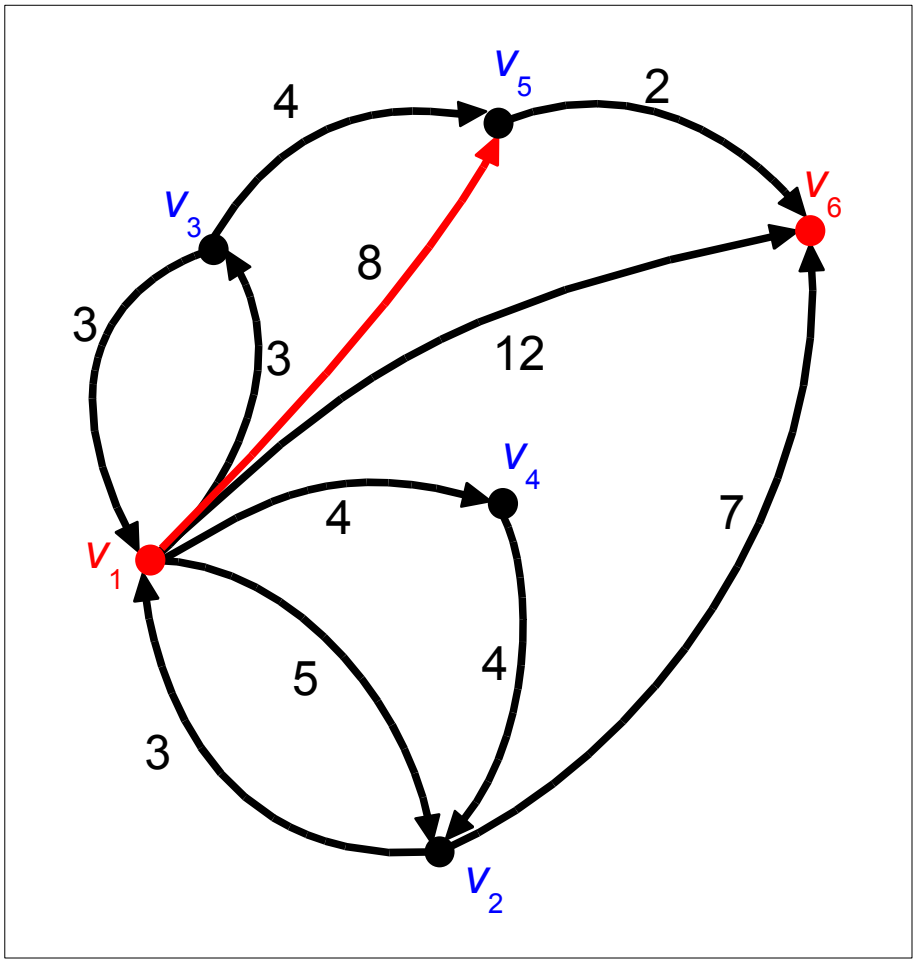
Krok	v_1	v_2	v_3	v_4	v_5	v_6
1	<u>0</u>	5	3	∞	∞	∞

Algorytm DIJKSTRY



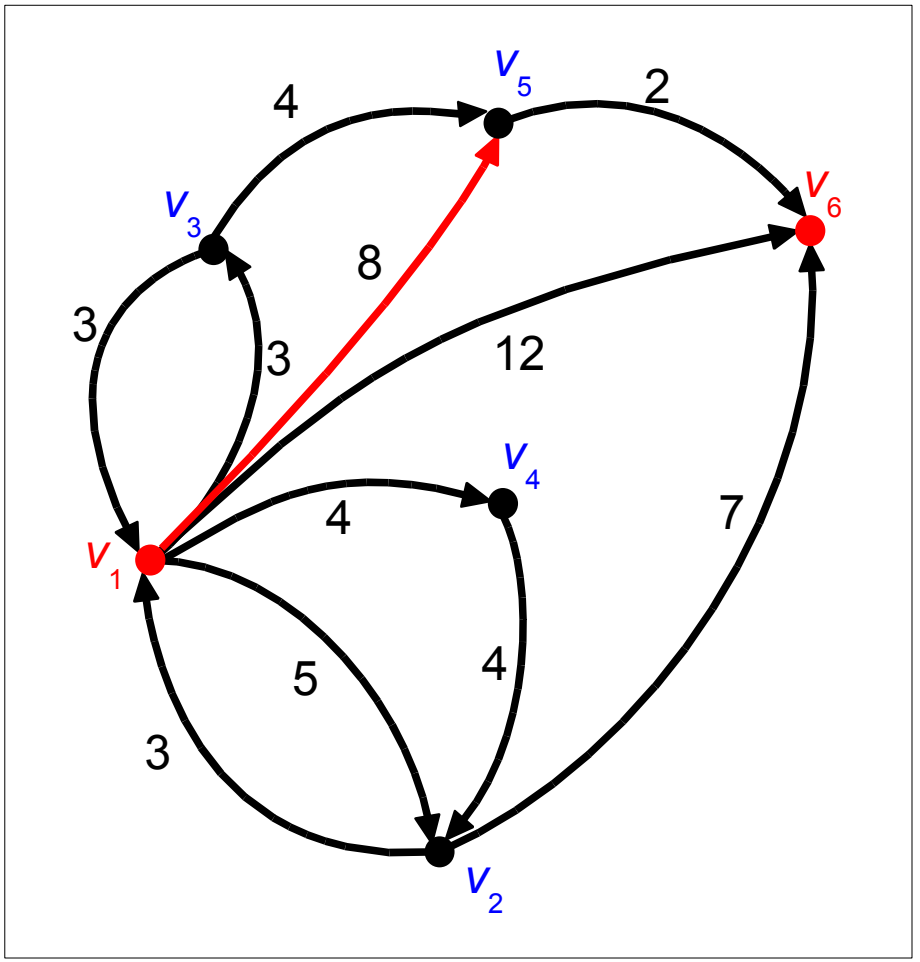
Krok	v_1	v_2	v_3	v_4	v_5	v_6
1	<u>0</u>	5	3	4	∞	∞

Algorytm DIJKSTRY



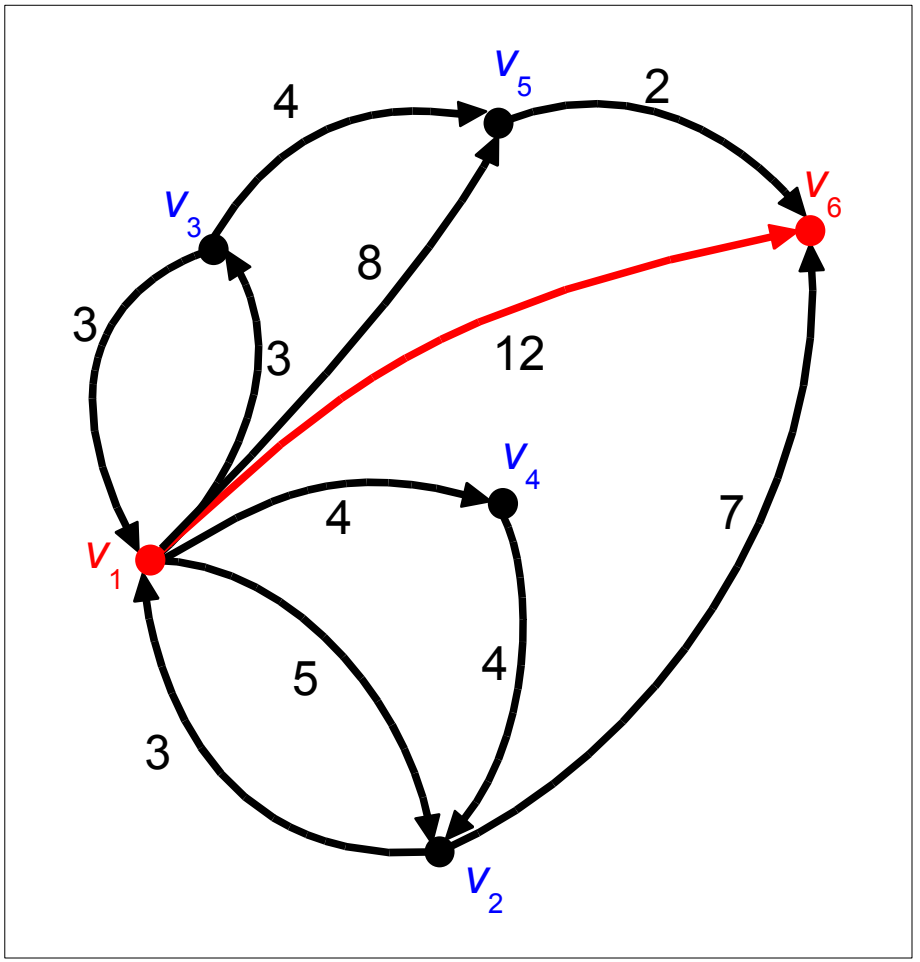
Krok	v_1	v_2	v_3	v_4	v_5	v_6
1	<u>0</u>	5	3	4	∞	∞

Algorytm DIJKSTRY



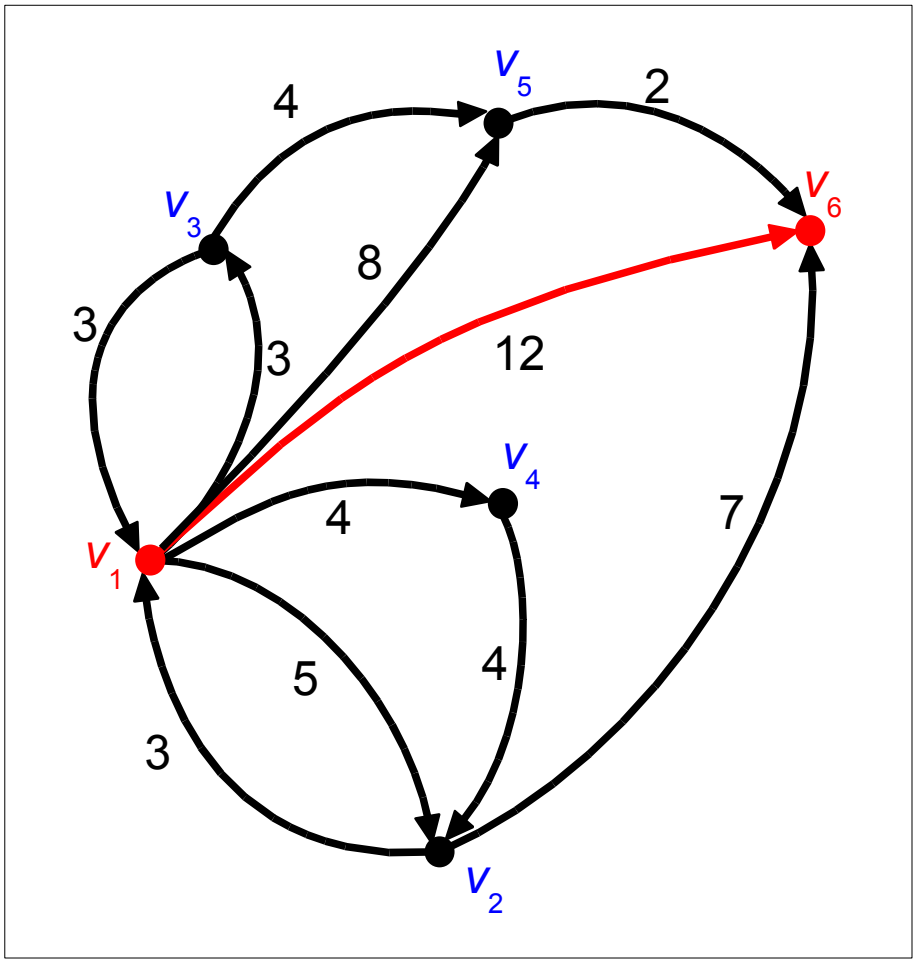
Krok	v_1	v_2	v_3	v_4	v_5	v_6
1	<u>0</u>	5	3	4	8	∞

Algorytm DIJKSTRY



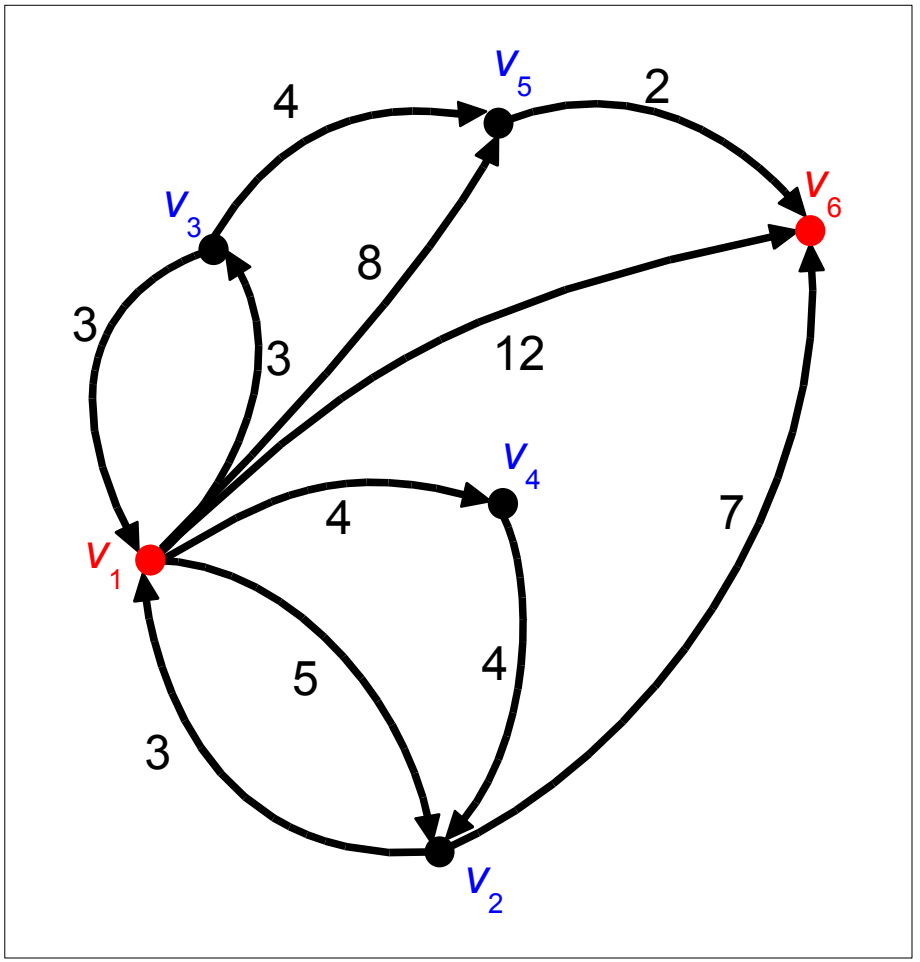
Krok	v_1	v_2	v_3	v_4	v_5	v_6
1	<u>0</u>	5	3	4	8	∞

Algorytm DIJKSTRY



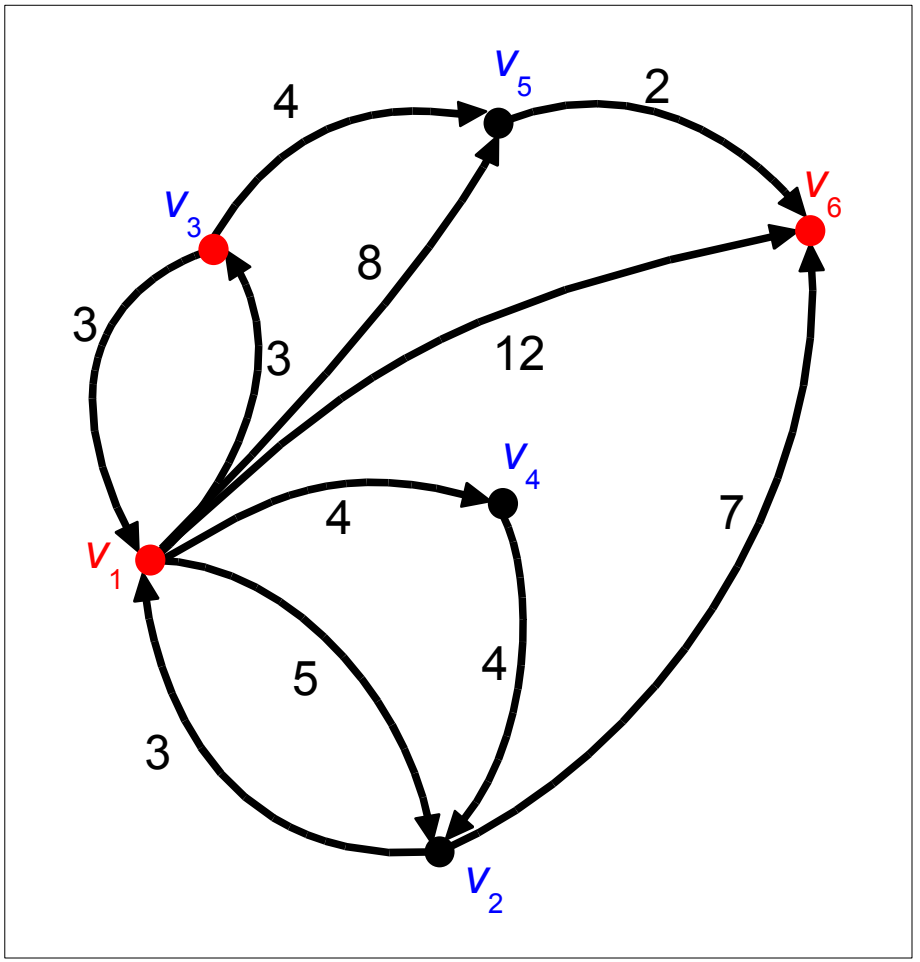
Krok	v_1	v_2	v_3	v_4	v_5	v_6
1	<u>0</u>	5	3	4	8	12

Algorytm DIJKSTRY



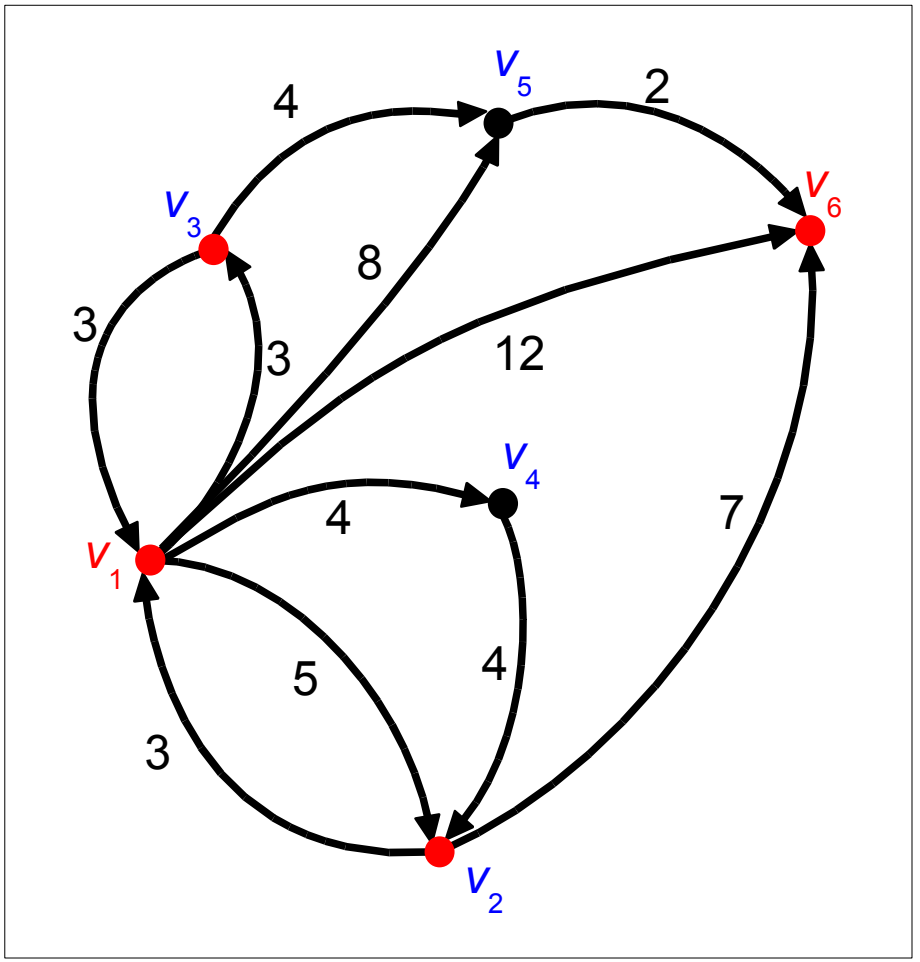
Krok	v_1	v_2	v_3	v_4	v_5	v_6
1	<u>0</u>	5	3	4	8	12
2	<u>0</u>	5	3	4	8	12

Algorytm DIJKSTRY



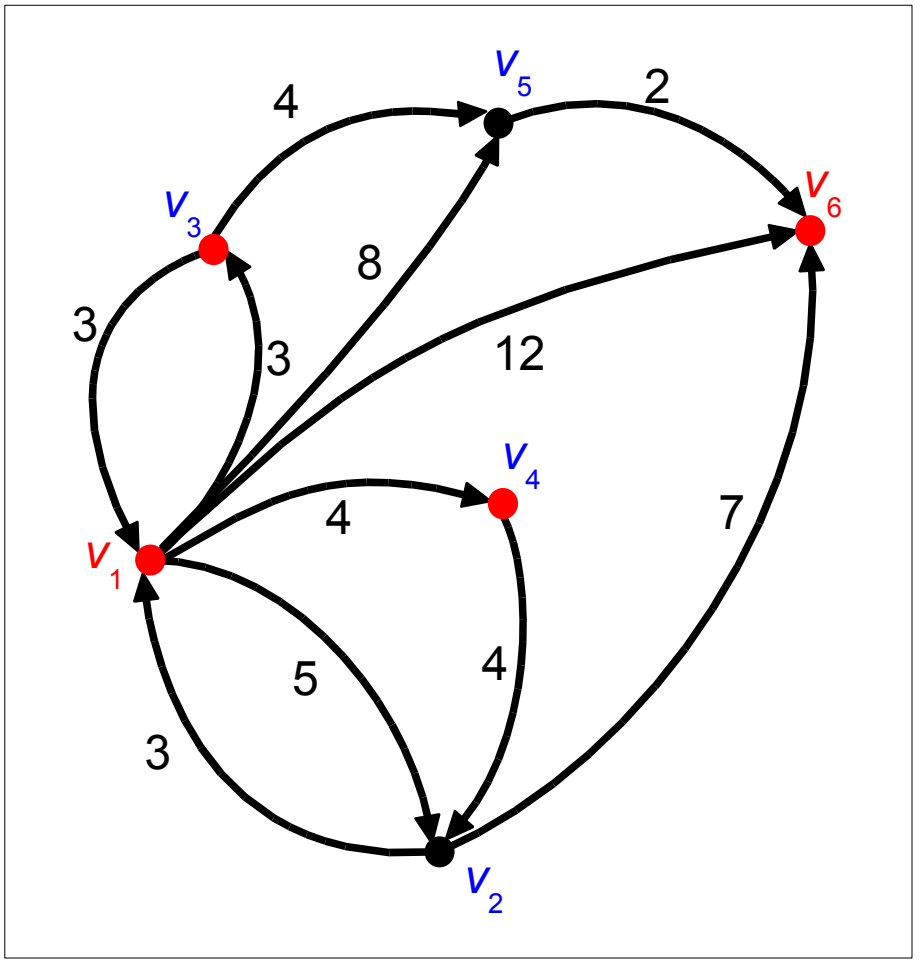
Krok	v_1	v_2	v_3	v_4	v_5	v_6
1	<u>0</u>	5	3	4	8	12
2	<u>0</u>	5	<u>3</u>	4	8	12

Algorytm DIJKSTRY



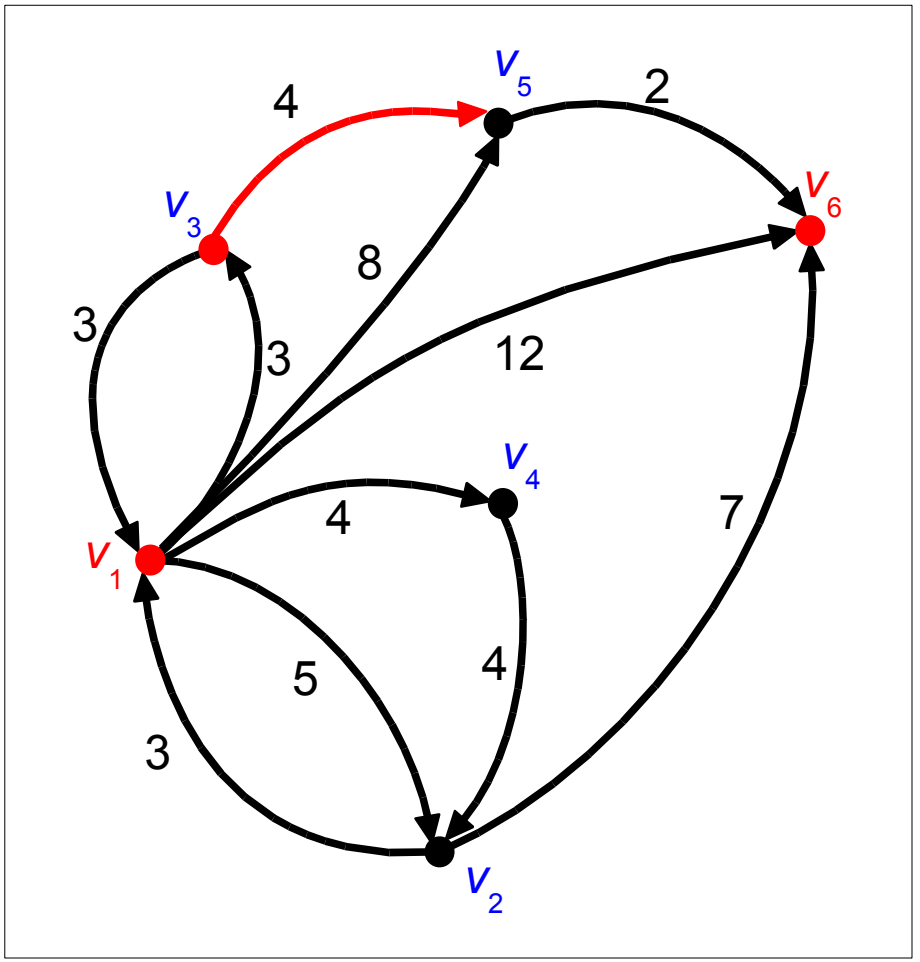
Krok	v_1	v_2	v_3	v_4	v_5	v_6
1	0	5	3	4	8	12
2	0	5	3	4	8	12

Algorytm DIJKSTRY



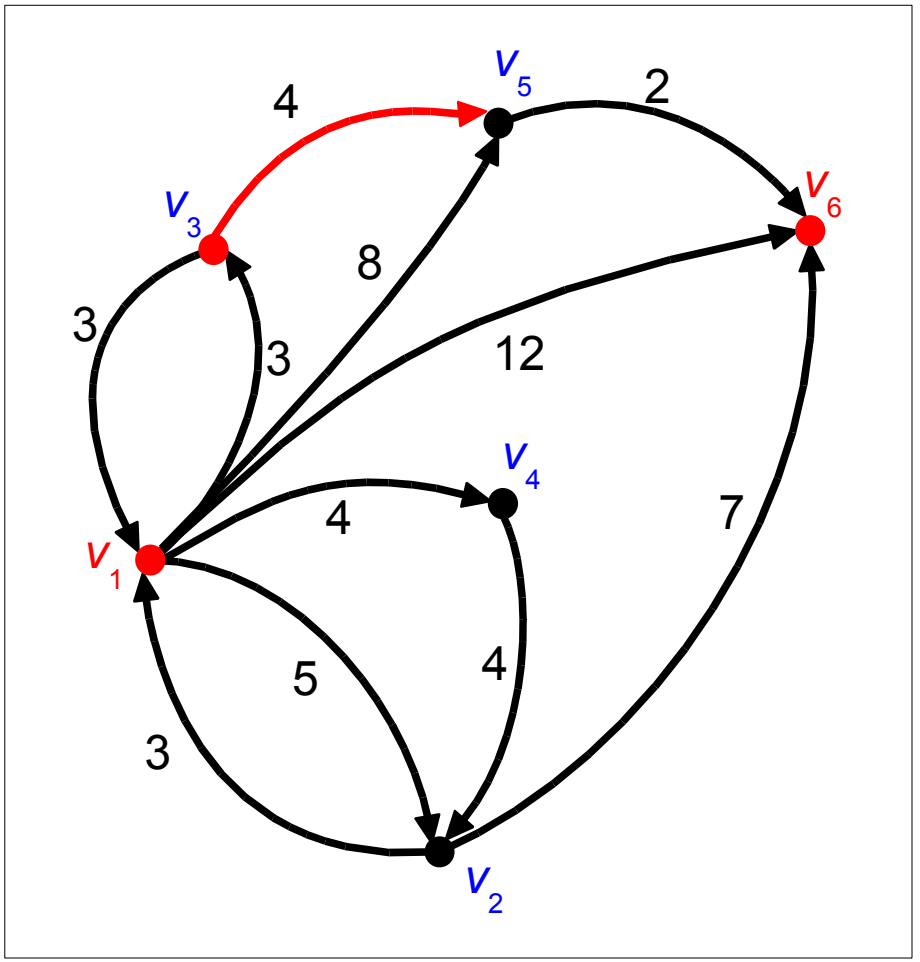
Krok	v_1	v_2	v_3	v_4	v_5	v_6
1	0	5	3	4	8	12
2	0	5	3	4	8	12

Algorytm DIJKSTRY



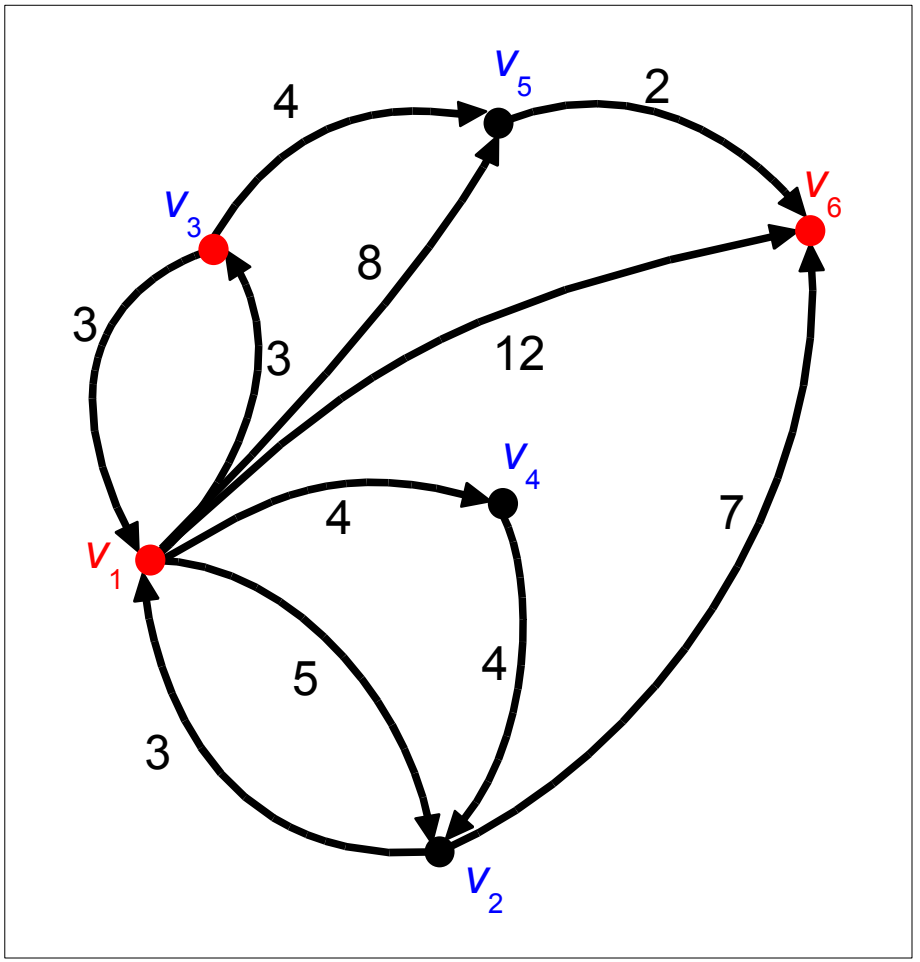
Krok	v_1	v_2	v_3	v_4	v_5	v_6
1	0	5	3	4	8	12
2	0	5	3	4	8	12

Algorytm DIJKSTRY



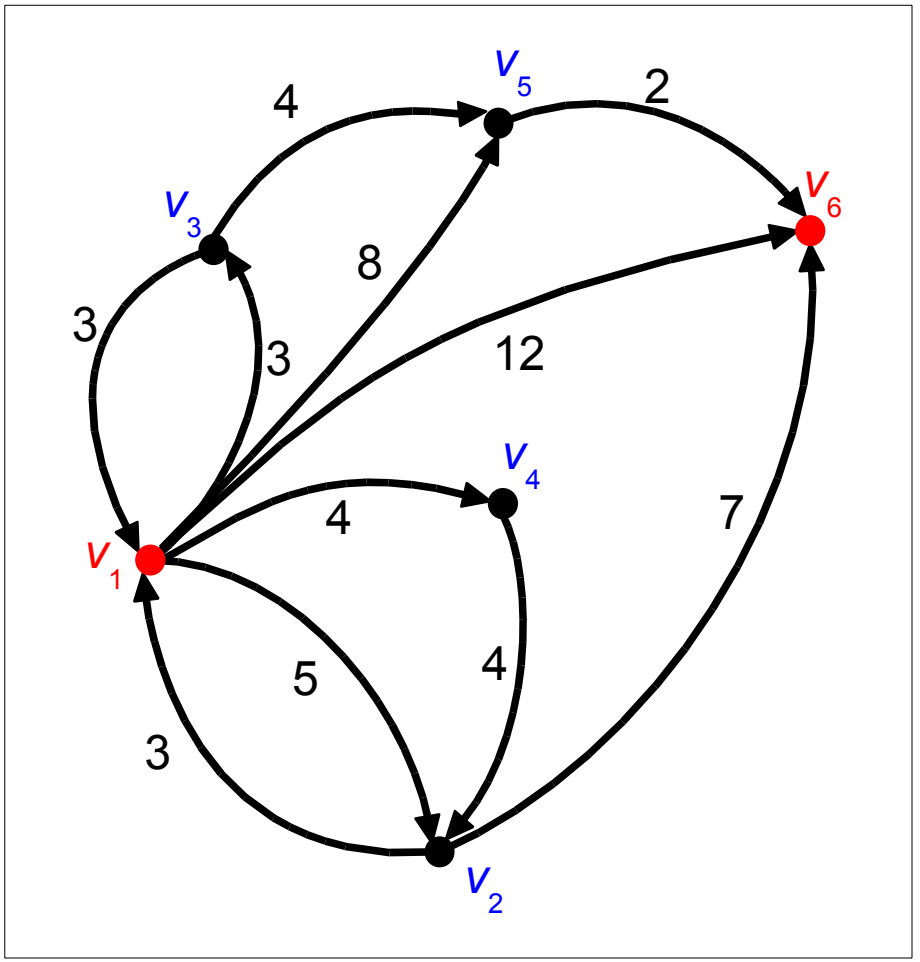
Krok	v_1	v_2	v_3	v_4	v_5	v_6
1	0	5	3	4	8	12
2	0	5	3	4	7	12

Algorytm DIJKSTRY



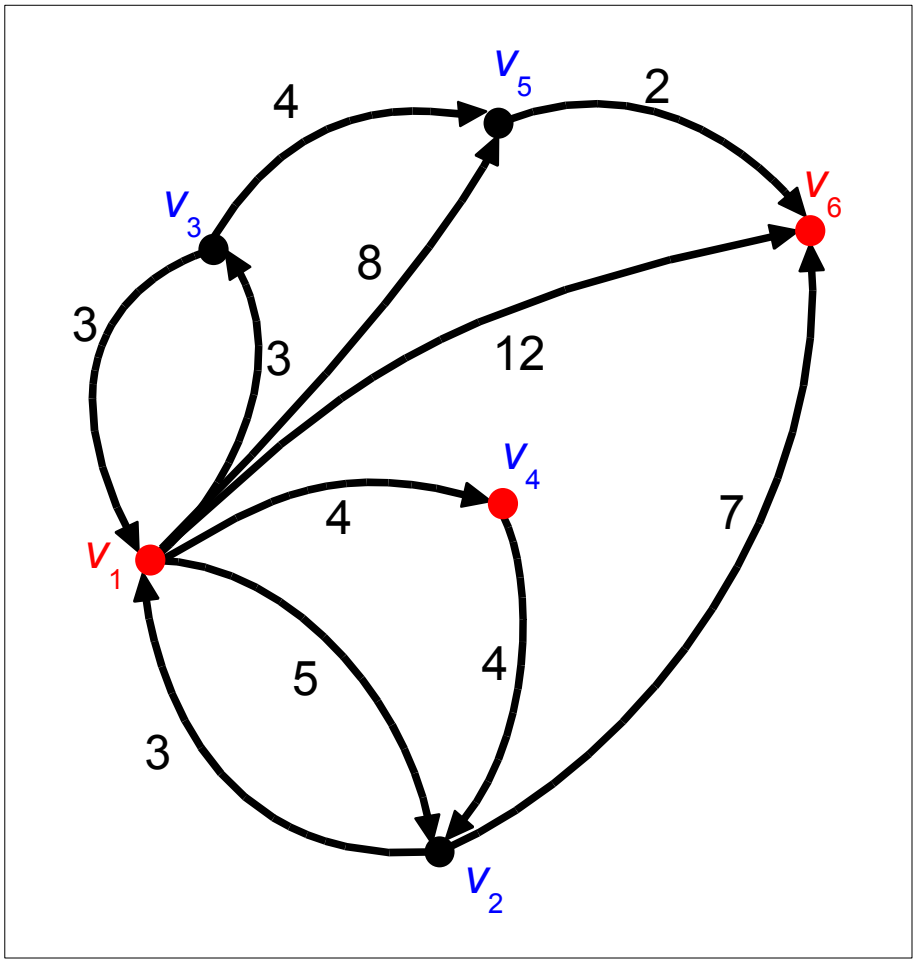
Krok	v_1	v_2	v_3	v_4	v_5	v_6
1	0	5	3	4	8	12
2	0	5	3	4	7	12

Algorytm DIJKSTRY



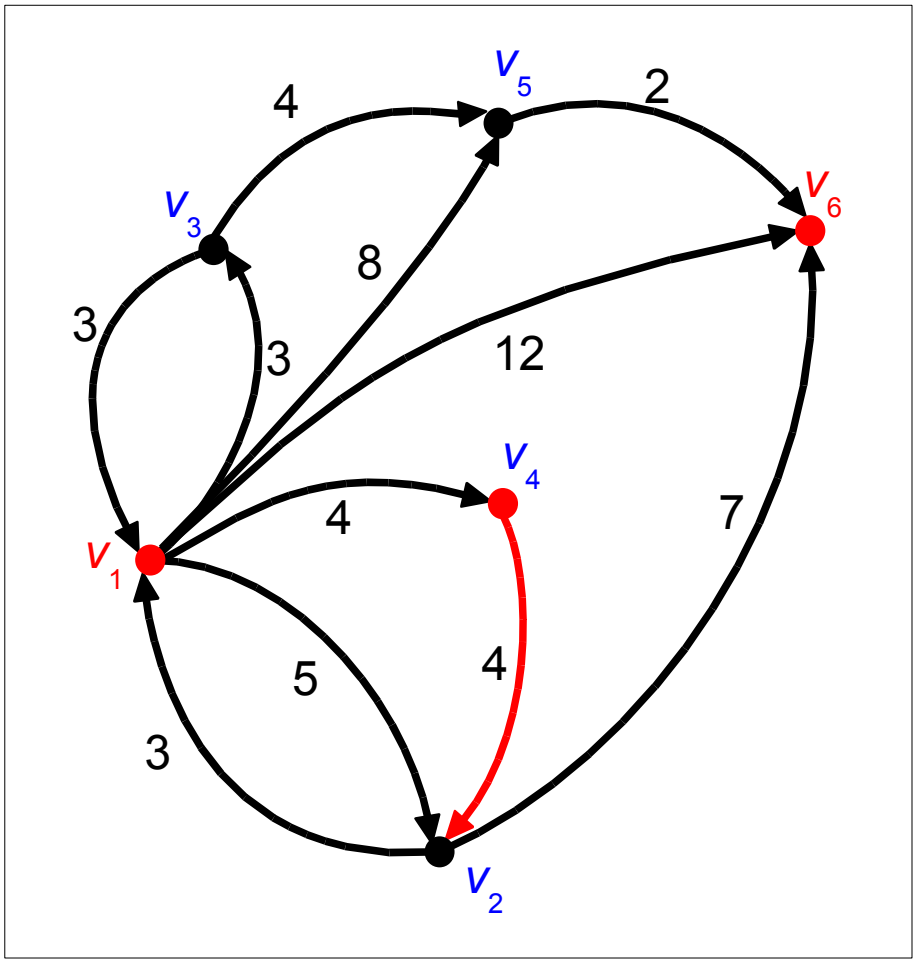
Krok	v_1	v_2	v_3	v_4	v_5	v_6
1	<u>0</u>	5	3	4	8	12
2	<u>0</u>	5	<u>3</u>	4	7	12
3	<u>0</u>	5	<u>3</u>	4	7	12

Algorytm DIJKSTRY



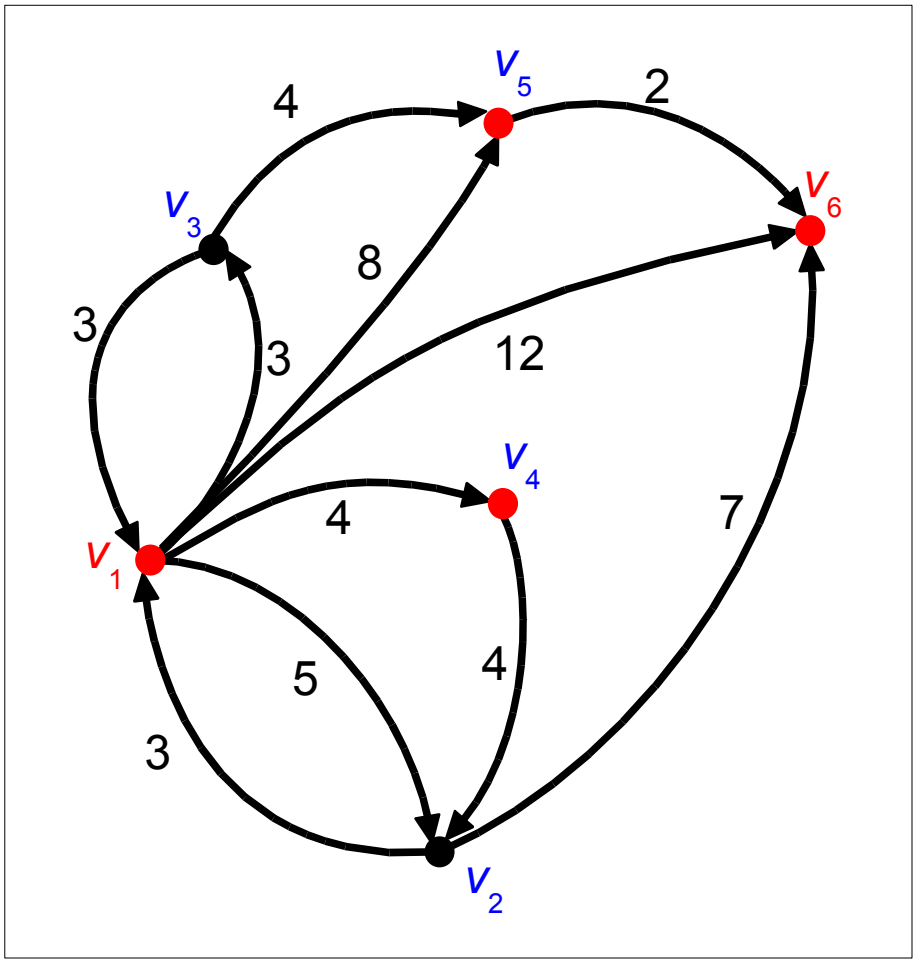
Krok	v_1	v_2	v_3	v_4	v_5	v_6
1	0	5	3	4	8	12
2	0	5	3	4	7	12
3	0	5	3	4	7	12

Algorytm DIJKSTRY



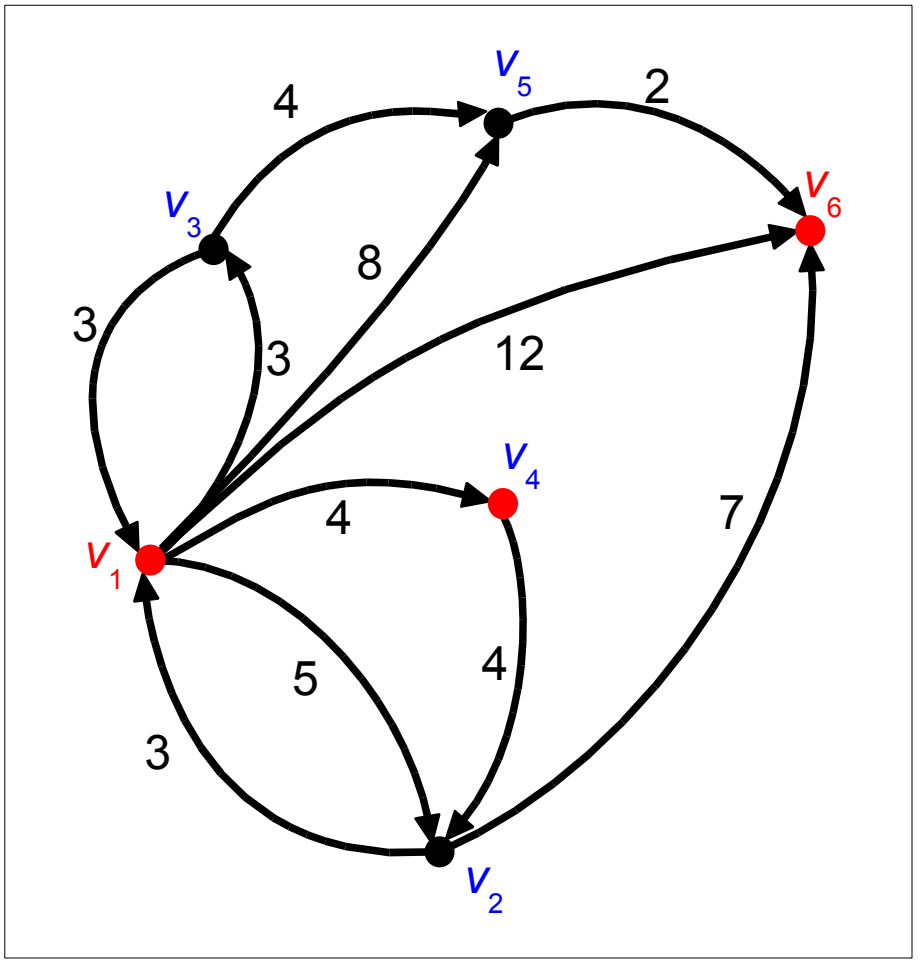
Krok	v_1	v_2	v_3	v_4	v_5	v_6
1	0	5	3	4	8	12
2	0	5	3	4	7	12
3	0	5	3	4	7	12

Algorytm DIJKSTRY



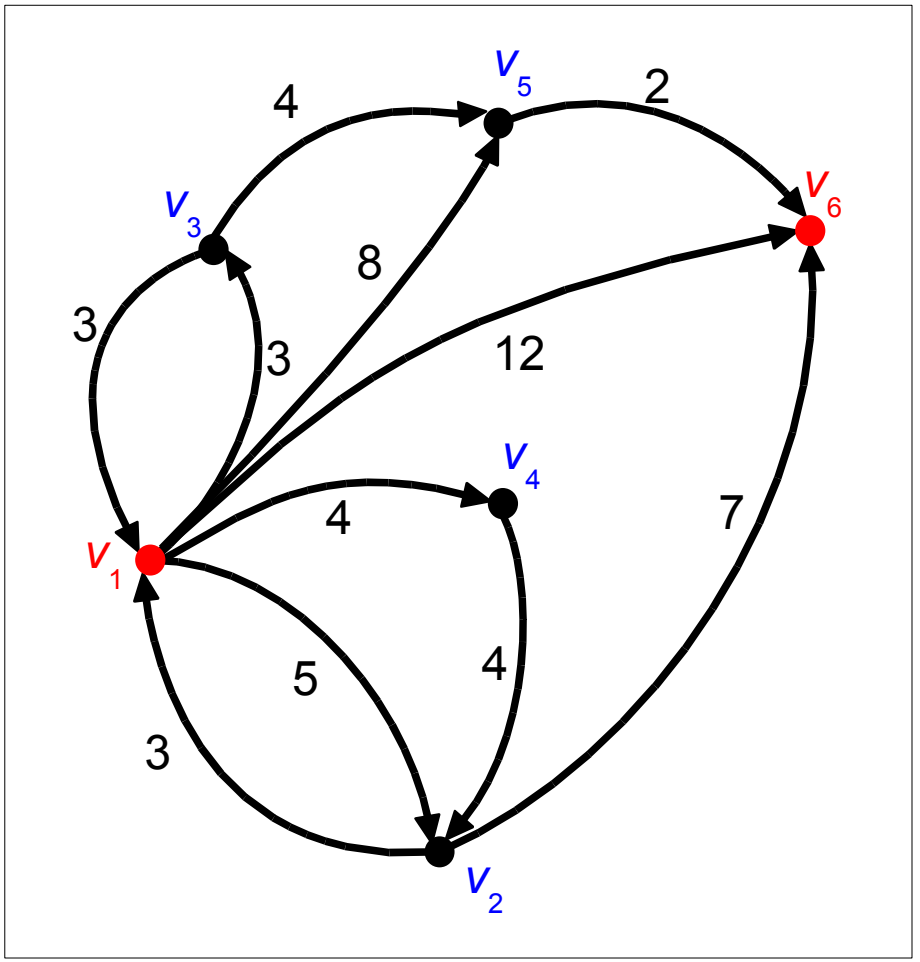
Krok	v_1	v_2	v_3	v_4	v_5	v_6
1	<u>0</u>	5	3	4	8	12
2	<u>0</u>	5	<u>3</u>	4	7	12
3	<u>0</u>	5	<u>3</u>	<u>4</u>	7	12

Algorytm DIJKSTRY



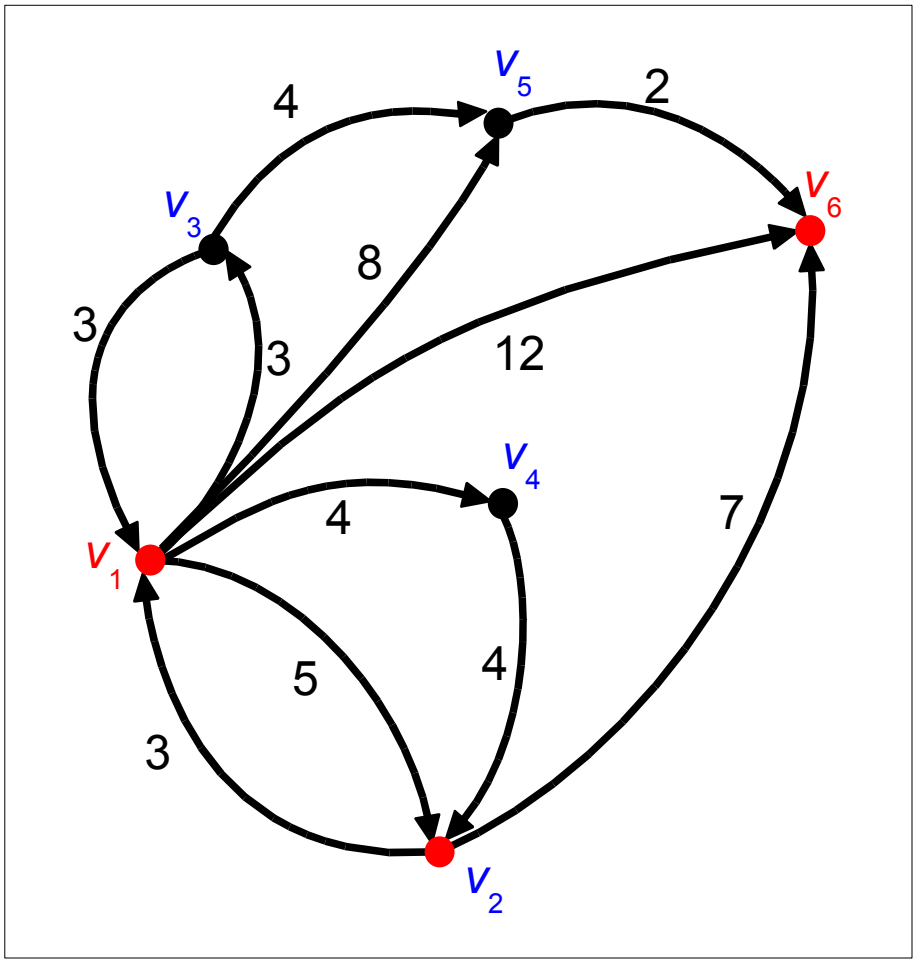
Krok	v_1	v_2	v_3	v_4	v_5	v_6
1	0	5	3	4	8	12
2	0	5	3	4	7	12
3	0	5	3	4	7	12

Algorytm DIJKSTRY



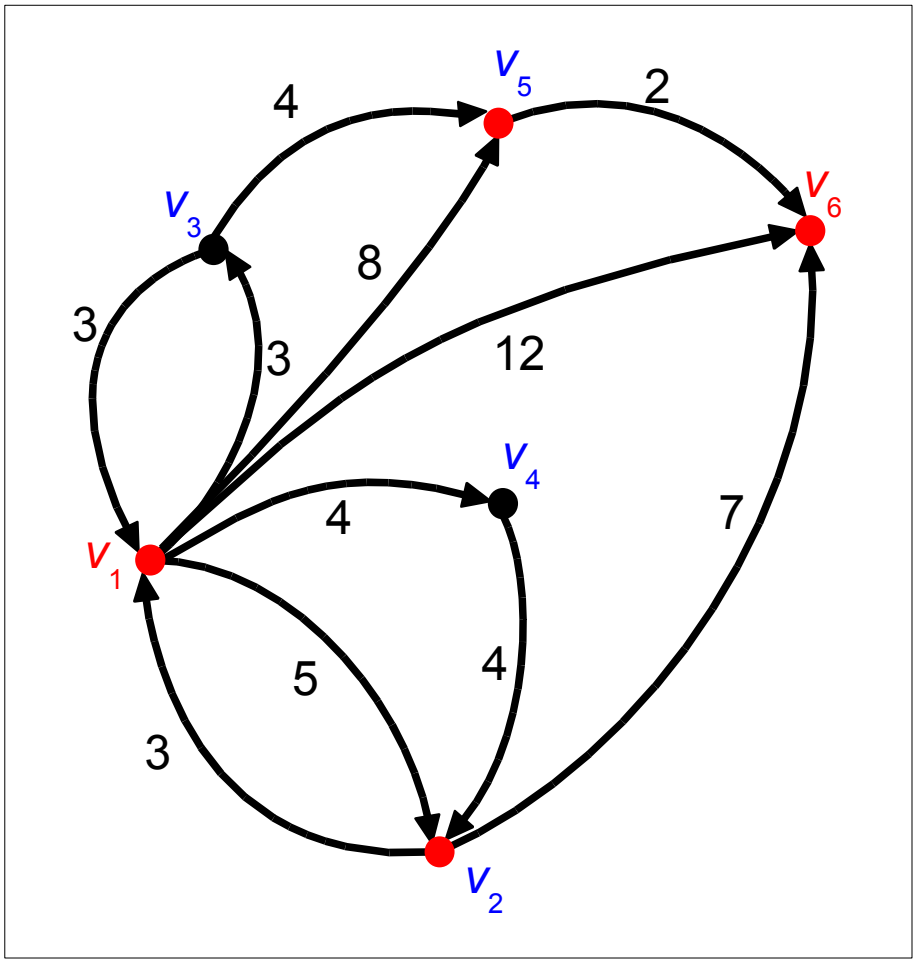
Krok	v_1	v_2	v_3	v_4	v_5	v_6
1	0	5	3	4	8	12
2	0	5	3	4	7	12
3	0	5	3	4	7	12
4	0	5	3	4	7	12

Algorytm DIJKSTRY



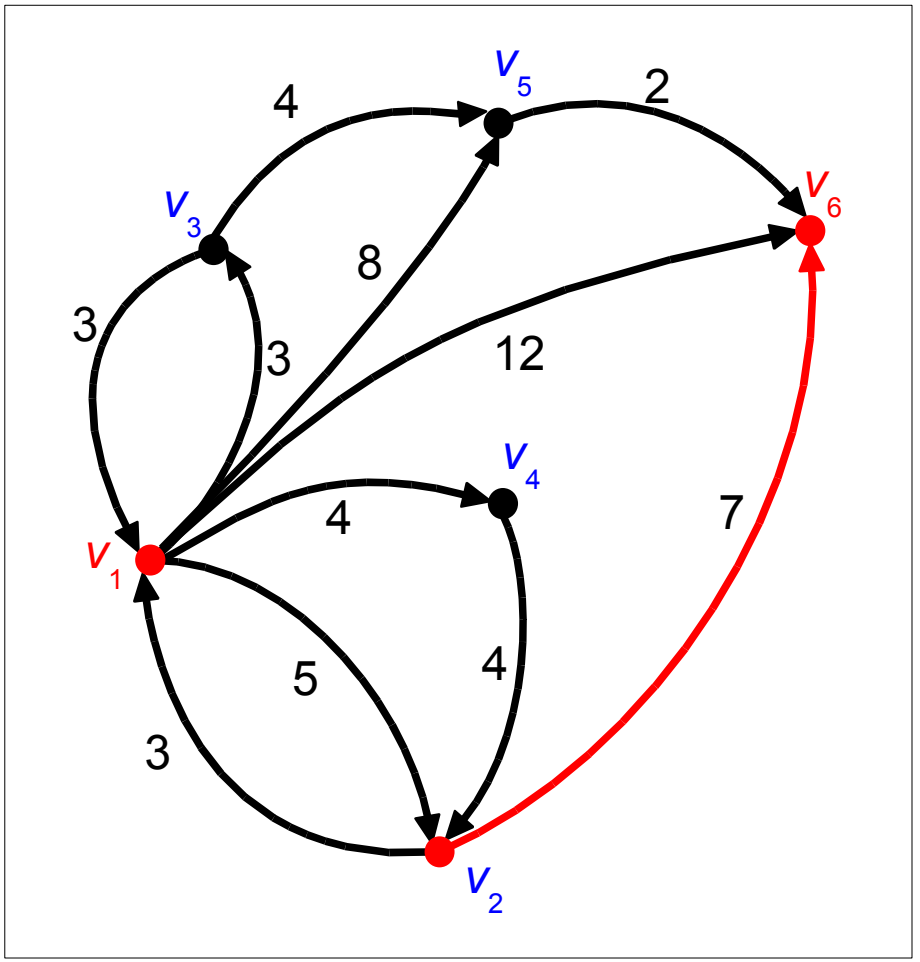
Krok	v_1	v_2	v_3	v_4	v_5	v_6
1	0	5	3	4	8	12
2	0	5	3	4	7	12
3	0	5	3	4	7	12
4	0	5	3	4	7	12

Algorytm DIJKSTRY



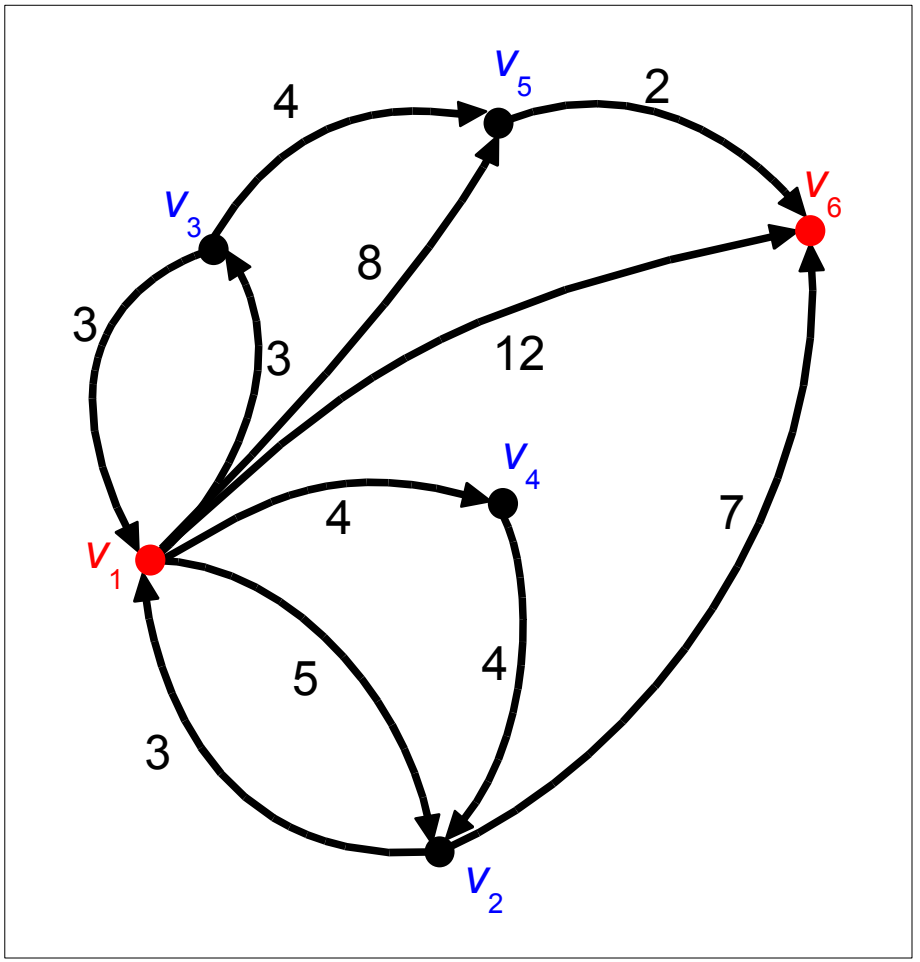
Krok	v_1	v_2	v_3	v_4	v_5	v_6
1	0	5	3	4	8	12
2	0	5	3	4	7	12
3	0	5	3	4	7	12
4	0	5	3	4	7	12

Algorytm DIJKSTRY



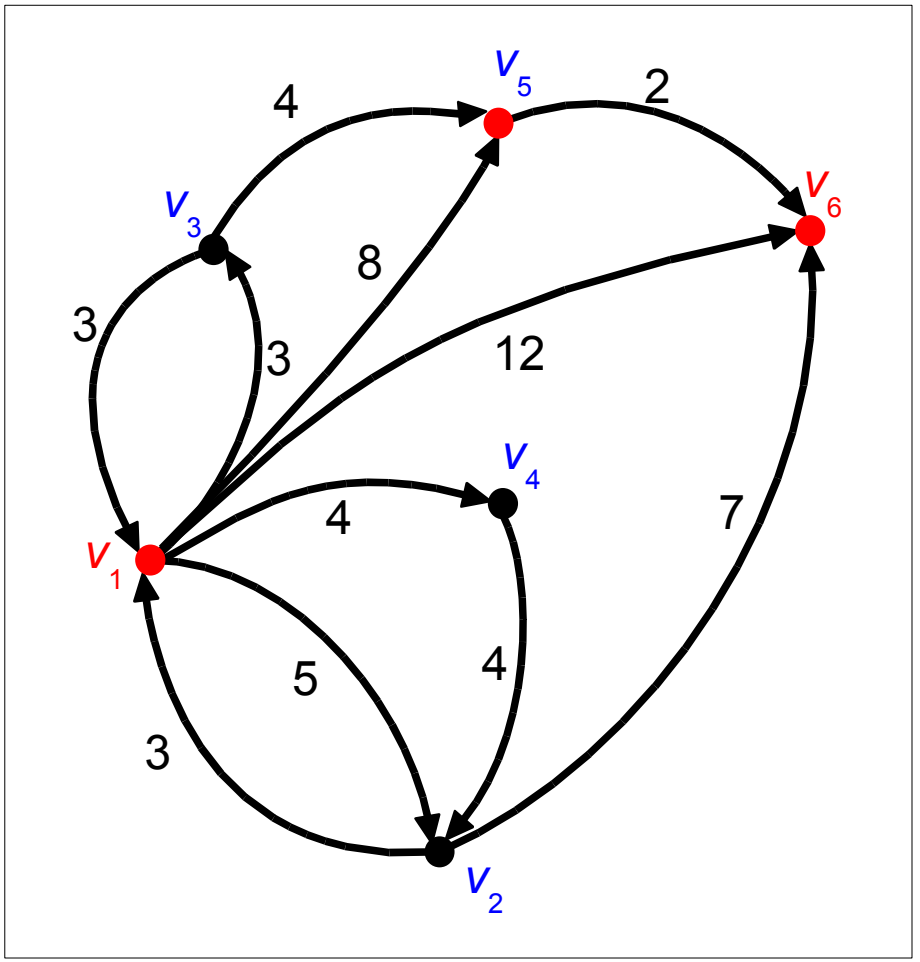
Krok	v_1	v_2	v_3	v_4	v_5	v_6
1	0	5	3	4	8	12
2	0	5	3	4	7	12
3	0	5	3	4	7	12
4	0	5	3	4	7	12

Algorytm DIJKSTRY



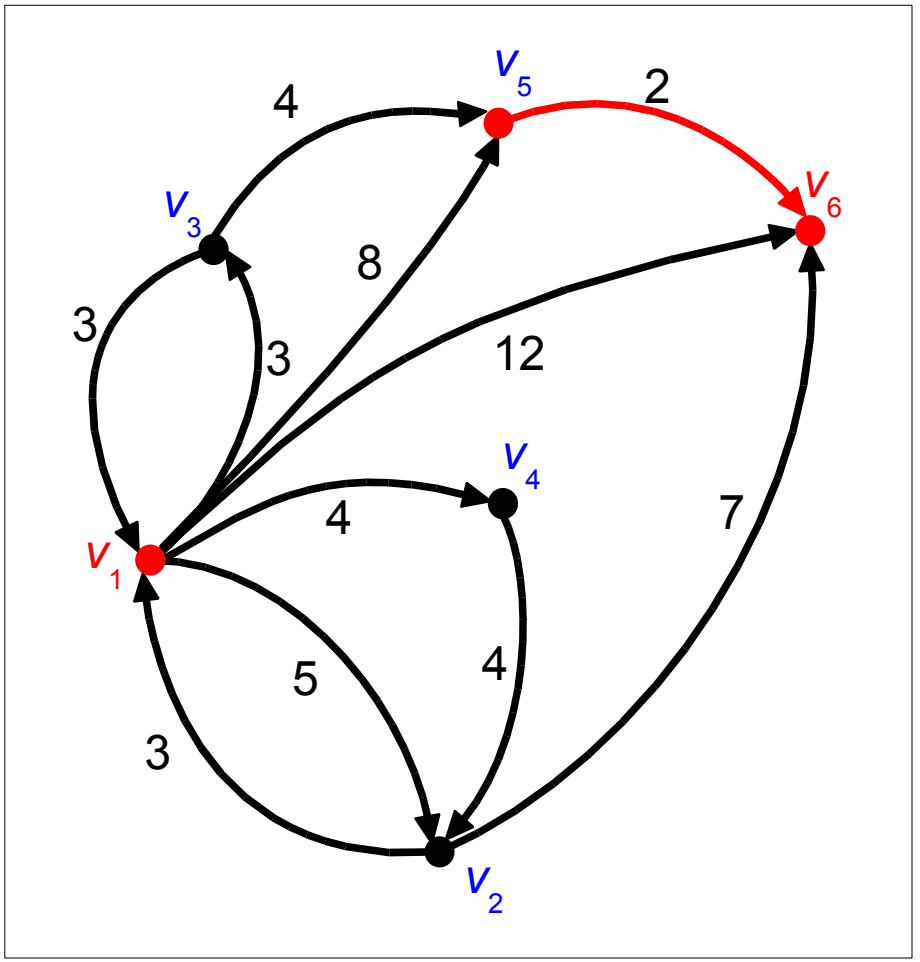
Krok	v_1	v_2	v_3	v_4	v_5	v_6
1	0	5	3	4	8	12
2	0	5	3	4	7	12
3	0	5	3	4	7	12
4	0	5	3	4	7	12
5	0	5	3	4	7	12

Algorytm DIJKSTRY



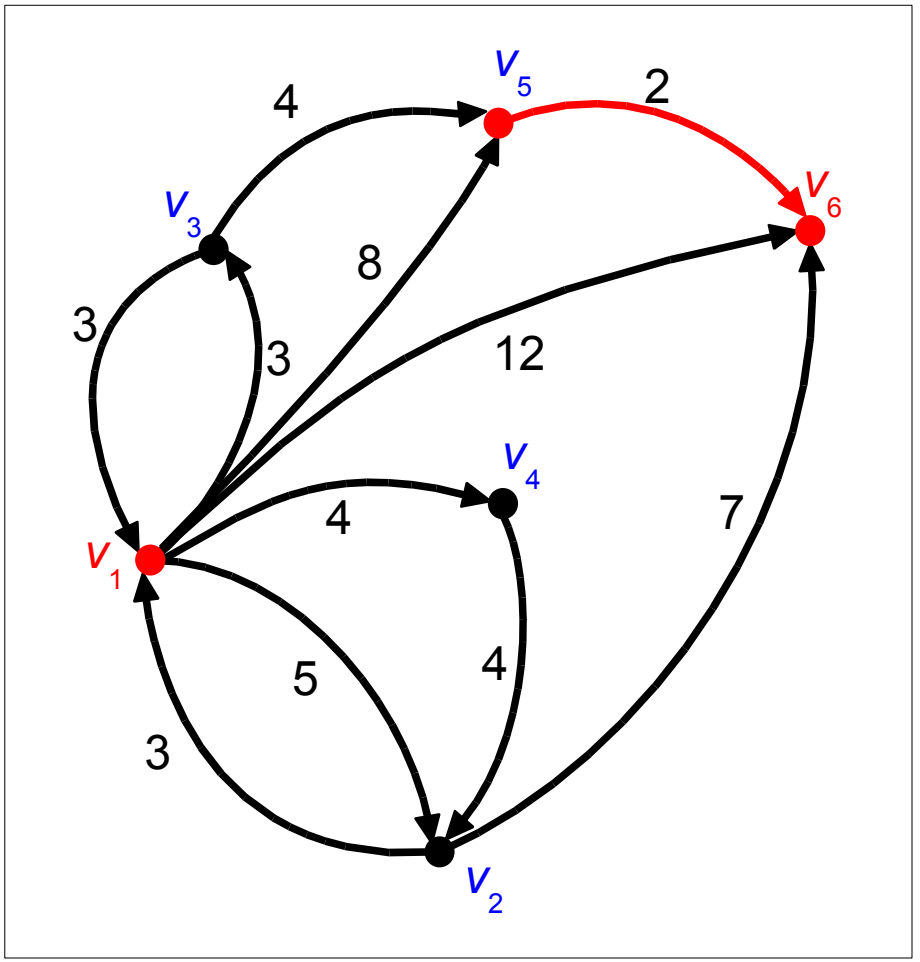
Krok	v_1	v_2	v_3	v_4	v_5	v_6
1	0	5	3	4	8	12
2	0	5	3	4	7	12
3	0	5	3	4	7	12
4	0	5	3	4	7	12
5	0	5	3	4	7	12

Algorytm DIJKSTRY



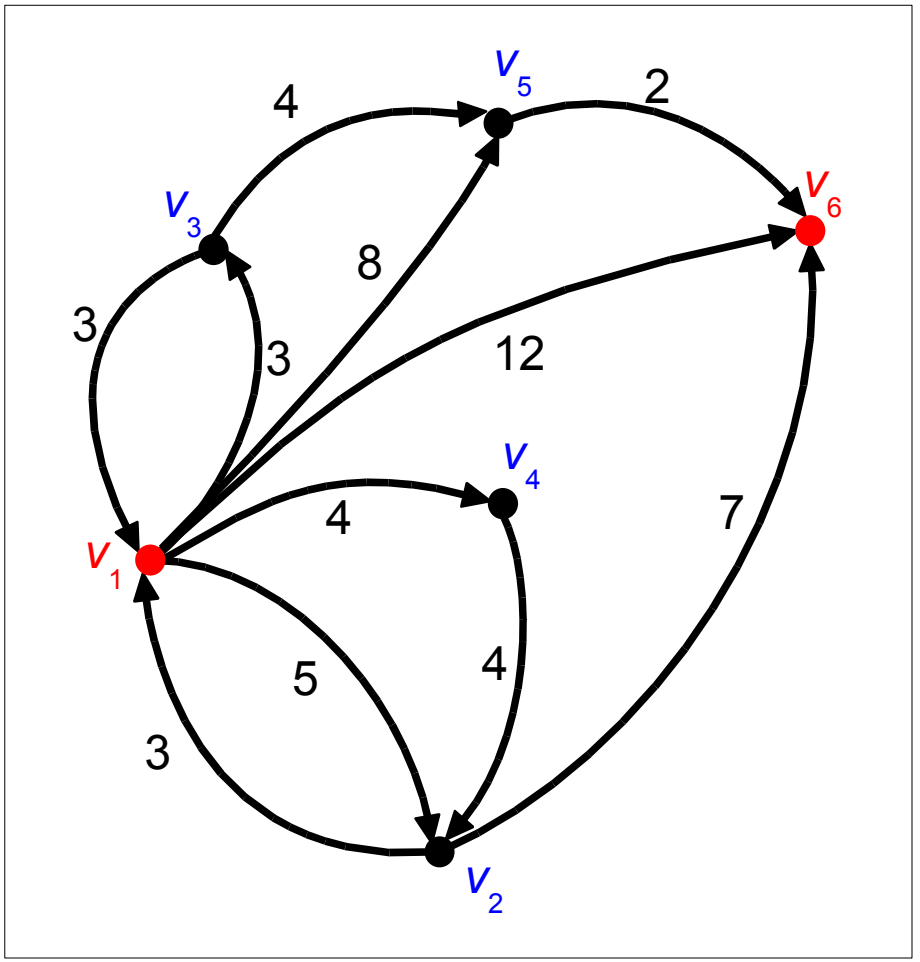
Krok	v_1	v_2	v_3	v_4	v_5	v_6
1	0	5	3	4	8	12
2	0	5	3	4	7	12
3	0	5	3	4	7	12
4	0	5	3	4	7	12
5	0	5	3	4	7	12

Algorytm DIJKSTRY



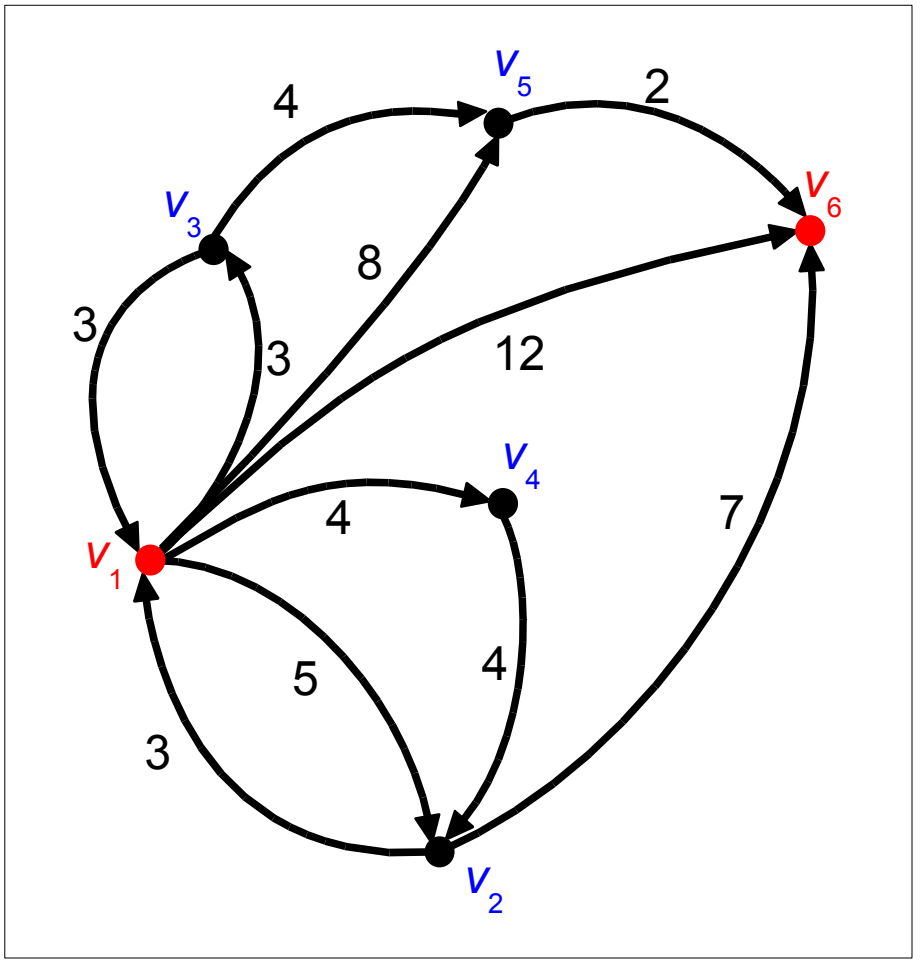
Krok	v_1	v_2	v_3	v_4	v_5	v_6
1	0	5	3	4	8	12
2	0	5	3	4	7	12
3	0	5	3	4	7	12
4	0	5	3	4	7	12
5	0	5	3	4	7	9

Algorytm DIJKSTRY



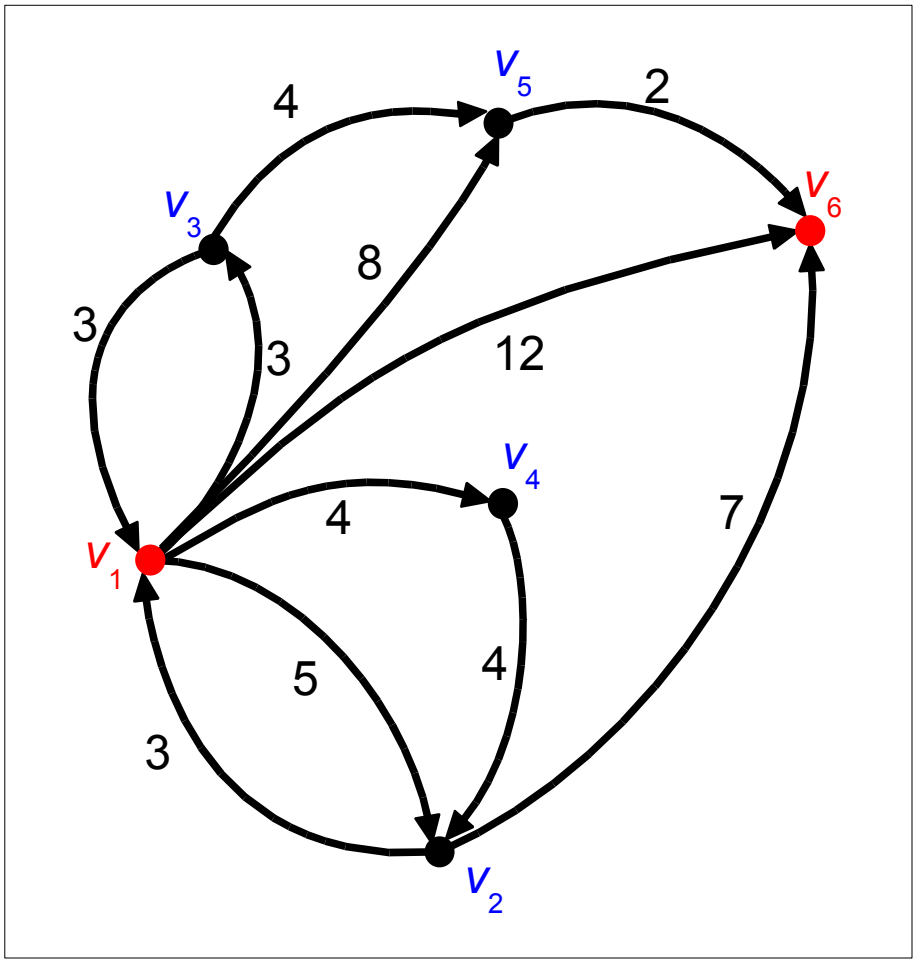
Krok	v_1	v_2	v_3	v_4	v_5	v_6
1	0	5	3	4	8	12
2	0	5	3	4	7	12
3	0	5	3	4	7	12
4	0	5	3	4	7	12
5	0	5	3	4	7	9
6	0	5	3	4	7	9

Algorytm DIJKSTRY



Krok	v_1	v_2	v_3	v_4	v_5	v_6
1	0	5	3	4	8	12
2	0	5	3	4	7	12
3	0	5	3	4	7	12
4	0	5	3	4	7	12
5	0	5	3	4	7	9
6	0	5	3	4	7	9

Algorytm DIJKSTRY

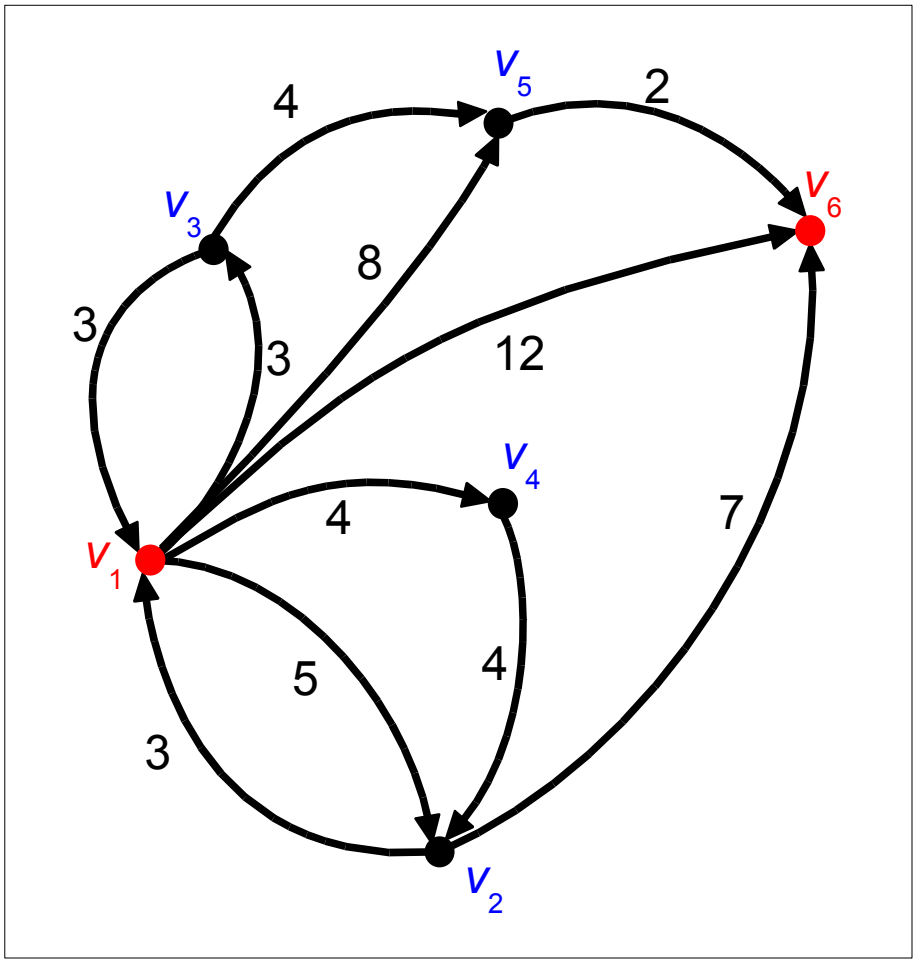


Krok	v_1	v_2	v_3	v_4	v_5	v_6
1	0	5	3	4	8	12
2	0	5	3	4	7	12
3	0	5	3	4	7	12
4	0	5	3	4	7	12
5	0	5	3	4	7	9
6	0	5	3	4	7	**9**

Najkrótsza droga z v_1 do v_6:

{ v_6}

Algorytm DIJKSTRY

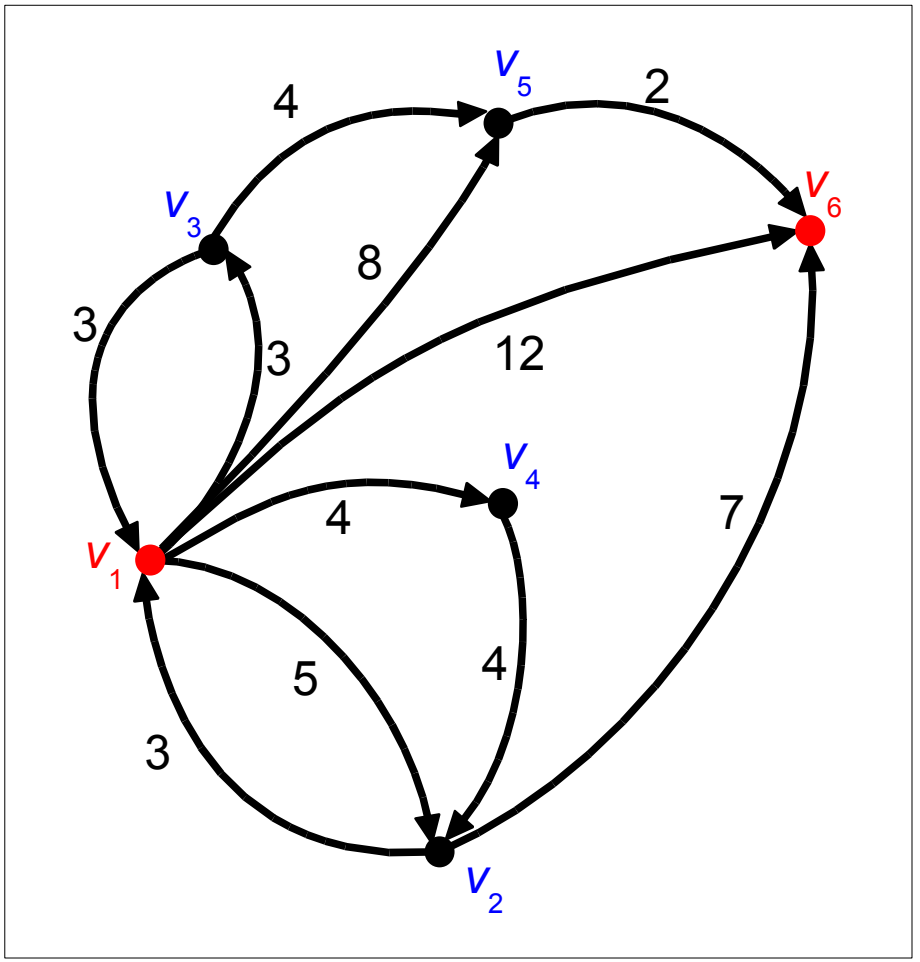


Krok	v_1	v_2	v_3	v_4	v_5	v_6
1	0	5	3	4	8	12
2	0	5	3	4	7	12
3	0	5	3	4	7	12
4	0	5	3	4	7	12
5	0	5	3	4	7	9
6	0	5	3	4	7	**9**

Najkrótsza droga z v_1 do v_6:

{ v_5, v_6}

Algorytm DIJKSTRY

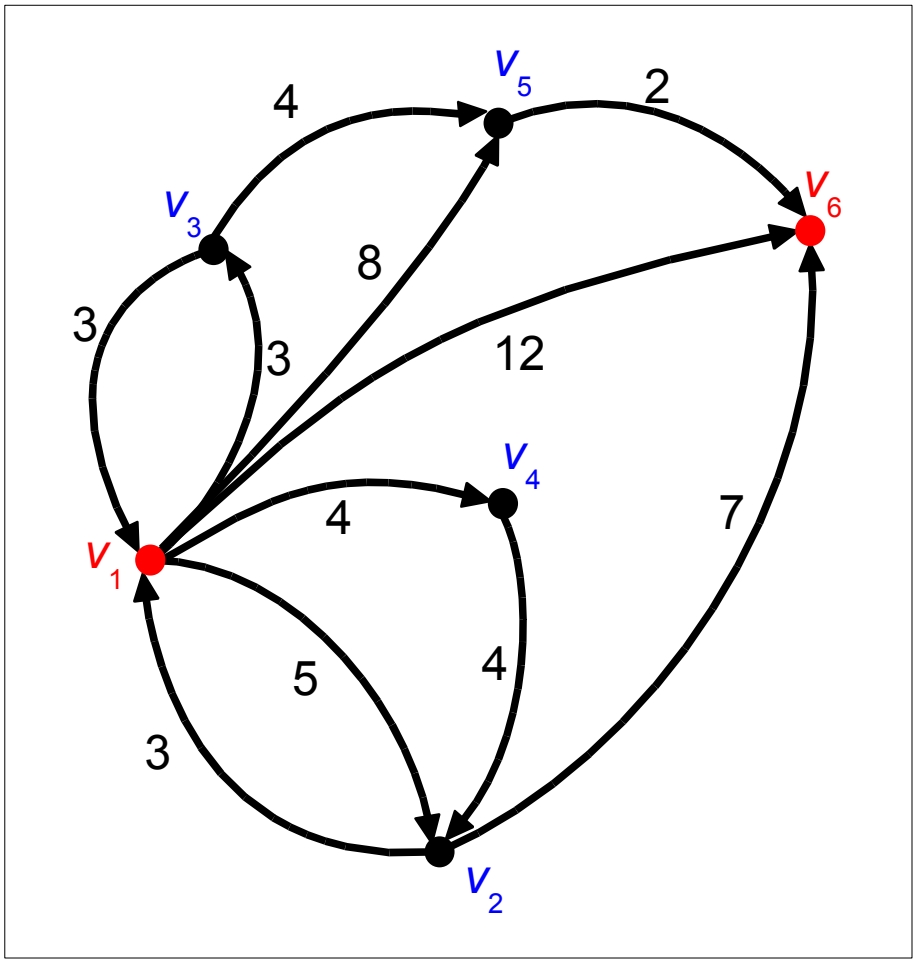


Krok	v_1	v_2	v_3	v_4	v_5	v_6
1	0	5	3	4	8	12
2	0	5	3	4	7	12
3	0	5	3	4	7	12
4	0	5	3	4	7	12
5	0	5	3	4	**7**	9
6	0	5	3	4	7	9

Najkrótsza droga z v_1 do v_6:

{ v_5, v_6}

Algorytm DIJKSTRY

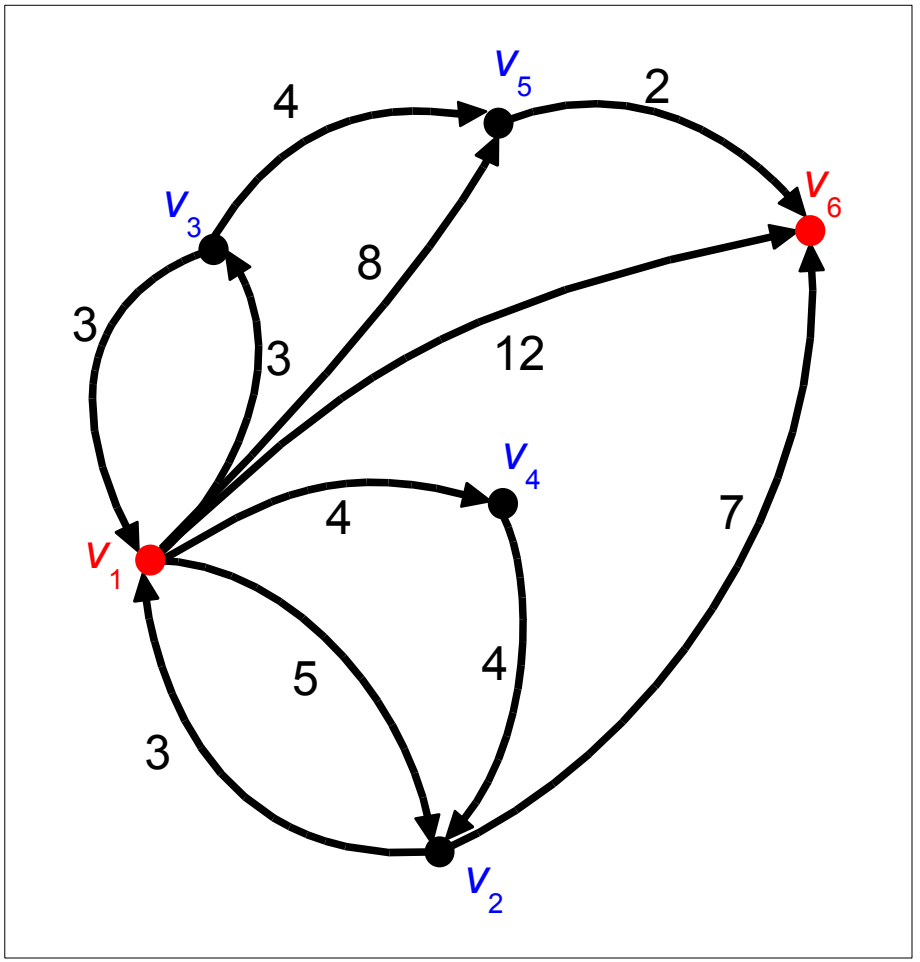


Krok	v_1	v_2	v_3	v_4	v_5	v_6
1	0	5	3	4	8	12
2	0	5	3	4	7	12
3	0	5	3	4	7	12
4	0	5	3	4	7	12
5	0	5	3	4	**7**	9
6	0	5	3	4	7	9

Najkrótsza droga z v_1 do v_6:

{ v_3, v_5, v_6}

Algorytm DIJKSTRY

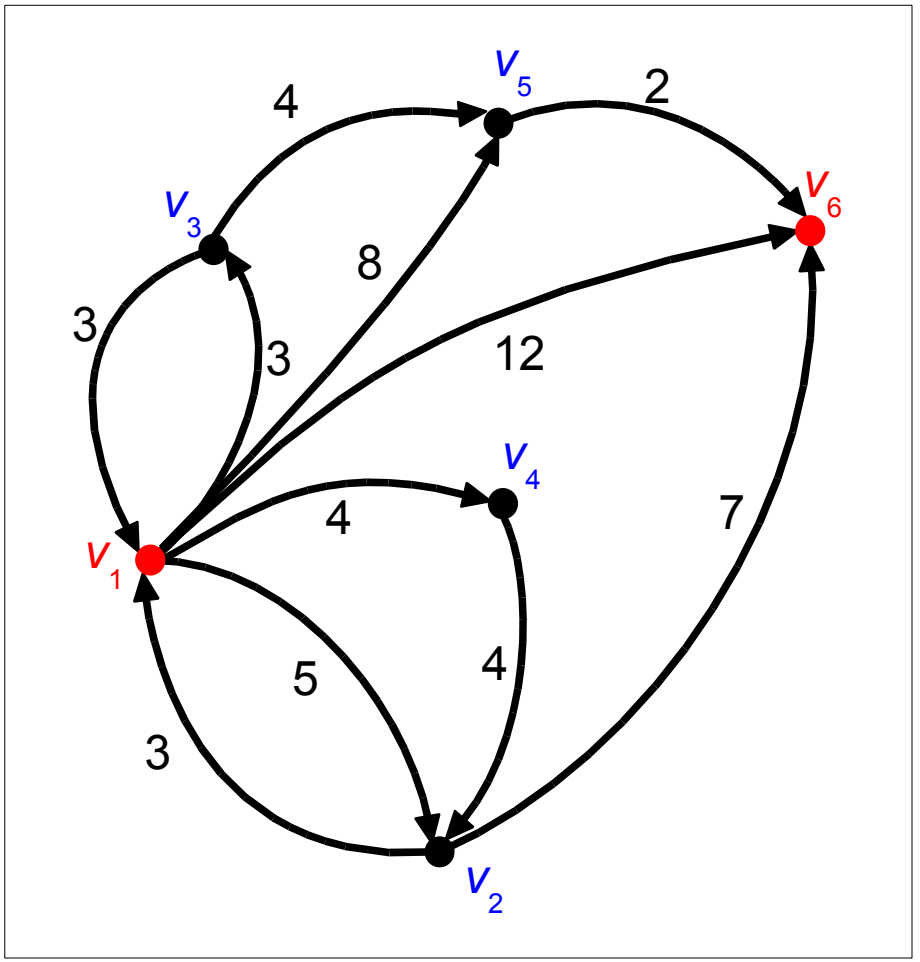


Krok	v_1	v_2	v_3	v_4	v_5	v_6
1	0	5	3	4	8	12
2	0	5	**3**	4	7	12
3	0	5	3	4	7	12
4	0	5	3	4	7	12
5	0	5	3	4	7	9
6	0	5	3	4	7	9

Najkrótsza droga z v_1 do v_6:

$\{ \quad v_3, v_5, v_6 \}$

Algorytm DIJKSTRY

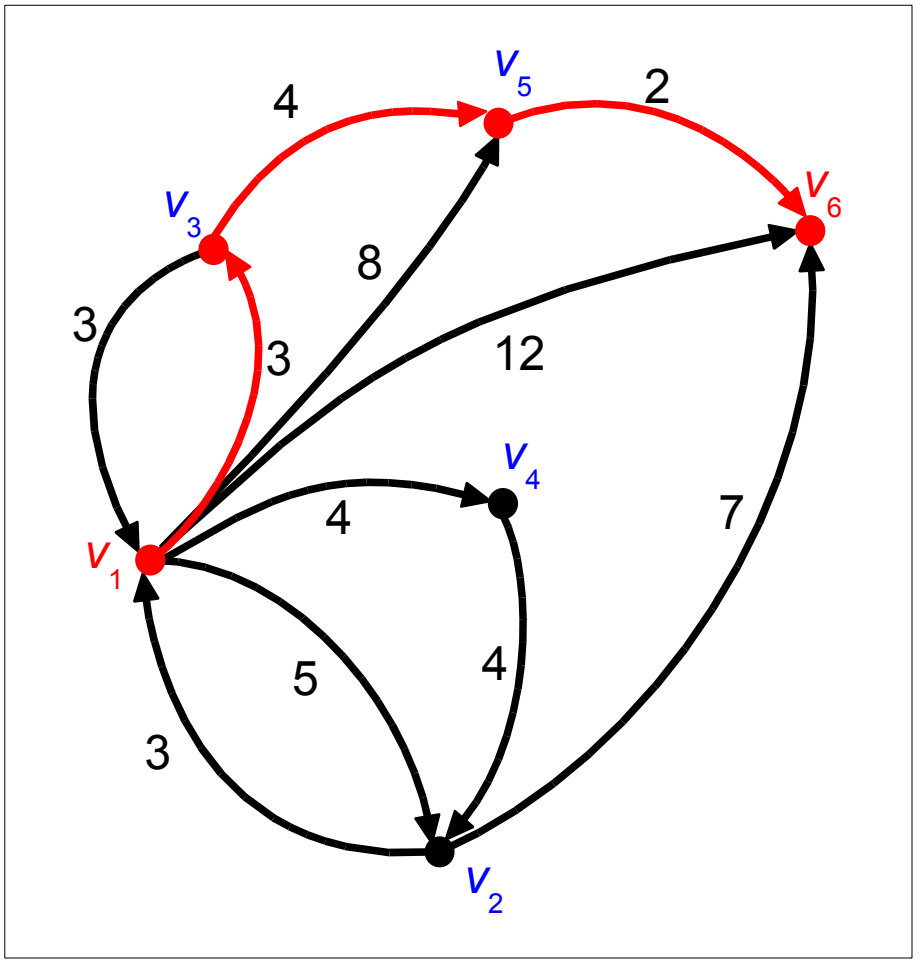


Krok	v_1	v_2	v_3	v_4	v_5	v_6
1	0	5	3	4	8	12
2	0	5	**3**	4	7	12
3	0	5	3	4	7	12
4	0	5	3	4	7	12
5	0	5	3	4	7	9
6	0	5	3	4	7	9

Najkrótsza droga z v_1 do v_6:

$\{ v_1, v_3, v_5, v_6 \}$

Algorytm DIJKSTRY



Krok	v_1	v_2	v_3	v_4	v_5	v_6
1	0	5	3	4	8	12
2	0	5	3	4	7	12
3	0	5	3	4	7	12
4	0	5	3	4	7	12
5	0	5	3	4	7	9
6	0	5	3	4	7	9

Najkrótsza droga z v_1 do v_6:

$\{ v_1, v_3, v_5, v_6 \}$

Złożoność obliczeniowa algorytmu Dijkstry:

Ponieważ w każdym kroku algorytmu ustalane jest oszacowanie jednego węzła, więc wszystkich iteracji jest n.

W każdej iteracji należy wyznaczyć węzeł o najmniejszym oszacowaniu – $O(\log n)$, jeśli oszacowania węzłów nieustalonych będziemy przechowywali w kopcu.

W tej samej iteracji należy jeszcze dokonać relaksacji łuków wychodzących z ustalonego – $O(n)$.

Zatem całkowita złożoność algorytmu wynosi $O(n^2)$. Możliwa jest jednak taka implementacja algorytmu, aby osiągnąć złożoność $O(m \log n)$.

Algorytm BELLMANA-FORDA

umożliwia, w przeciwieństwie do algorytmu Dijkstry, wyznaczanie najkrótszych dróg z wyznaczonego węzła do pozostałych przy dodatnich i **ujemnych wagach**. Warunkiem jest, aby graf **nie zawierał cykli o ujemnej wadze**, osiągalnych ze źródła.

W algorytmie tym również wykorzystuje się **relaksację** łuków wychodzących. Jednak żeby metodę tą można było z powodzeniem wykorzystać w grafach z ujemnymi wagami, należy w każdej z n iteracji dokonać relaksacji dla wszystkich m łuków.

Prowadzi to do **złożoności** $O(nm)$.

Zauważmy jednak, że **kolejność rozpatrywania łuków** do relaksacji może **wpływać na efektywność** algorytmu.

Np. w pierwszej iteracji zrelaksowanie najpierw łuków wychodzących ze źródła sprawi, że w tej samej iteracji będzie możliwe zrelaksowanie również łuków incydentnych do wcześniej wspomnianych, a potem incydentnych z tymi ostatnimi, itd.

Natomiast jeśli zaczniemy relaksację od łuków najbardziej oddalonych od źródła, to w pierwszej iteracji będziemy mogli zrelaksować jedynie łuki bezpośrednio wychodzące ze źródła (pozostałe zostaną zrelaksowane dopiero w kolejnych iteracjach).

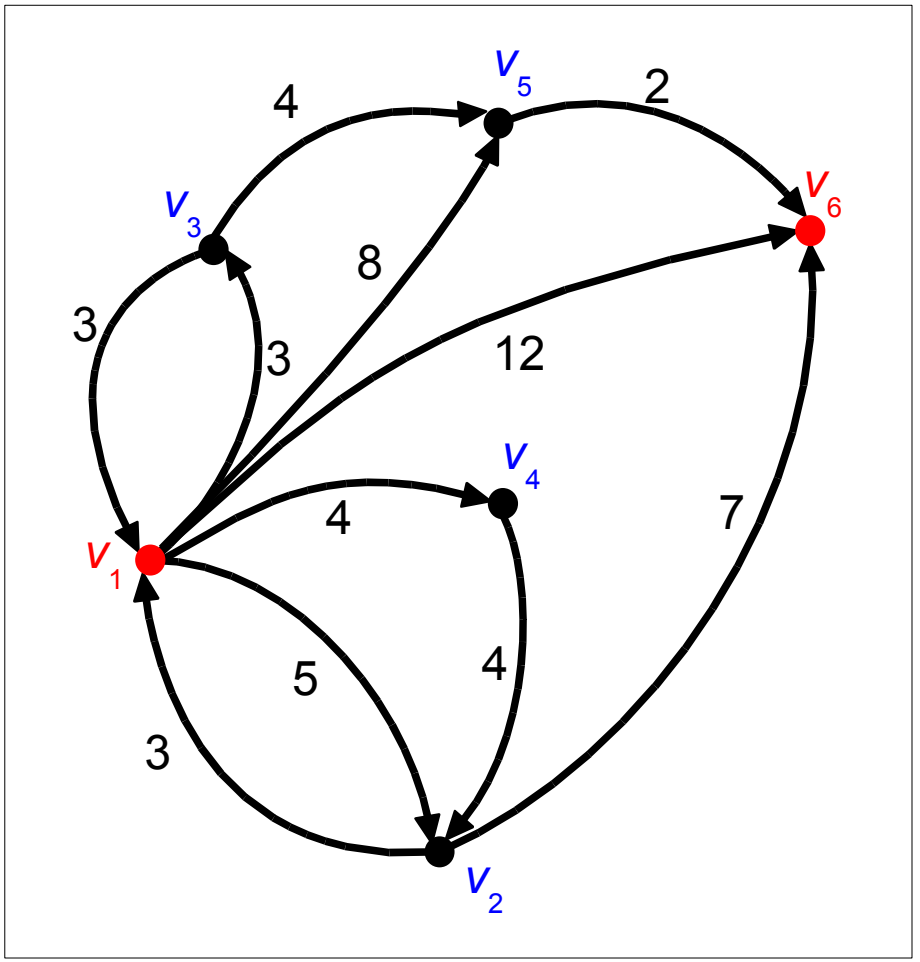
Oczywiście kolejność analizowania łuków **nie wpływa na optymalność** algorytmu – ta jest gwarantowana (teoretycznie udowodniona).

W związku z powyższym, ideę algorytmu Bellmana-Forda można zastosować do następującej implementacji.

- Tworzymy **kolejkę węzłów** do rozpatrzenia. Jej funkcja jest taka, że najpierw relaksujemy łuki wychodzące z pierwszego węzła z kolejki, potem z węzła drugiego, itd.
- W każdej iteracji **do kolejki** dołączamy węzły incydentne z rozpatrywanym, dla których nastąpiła poprawa i o ile już się tam nie znajdują.
- **Początkowo** kolejka zawiera tylko źródło, a algorytm **kończy się** gdy w kolejce nie będzie już żadnych węzłów.
- Dodatkowo, ważne jest **miejsce umieszczania** węzłów w kolejce:
  - jeśli węzeł już się znajduje w kolejce, to zostaje na tym miejscu;
  - jeśli był już wcześniej w kolejce (ale obecnie nie znajduje się w niej), to jest umieszczany na jej początku;
  - w przeciwnym wypadku trafia na koniec.

Algorytm BELLMANA-FORDA

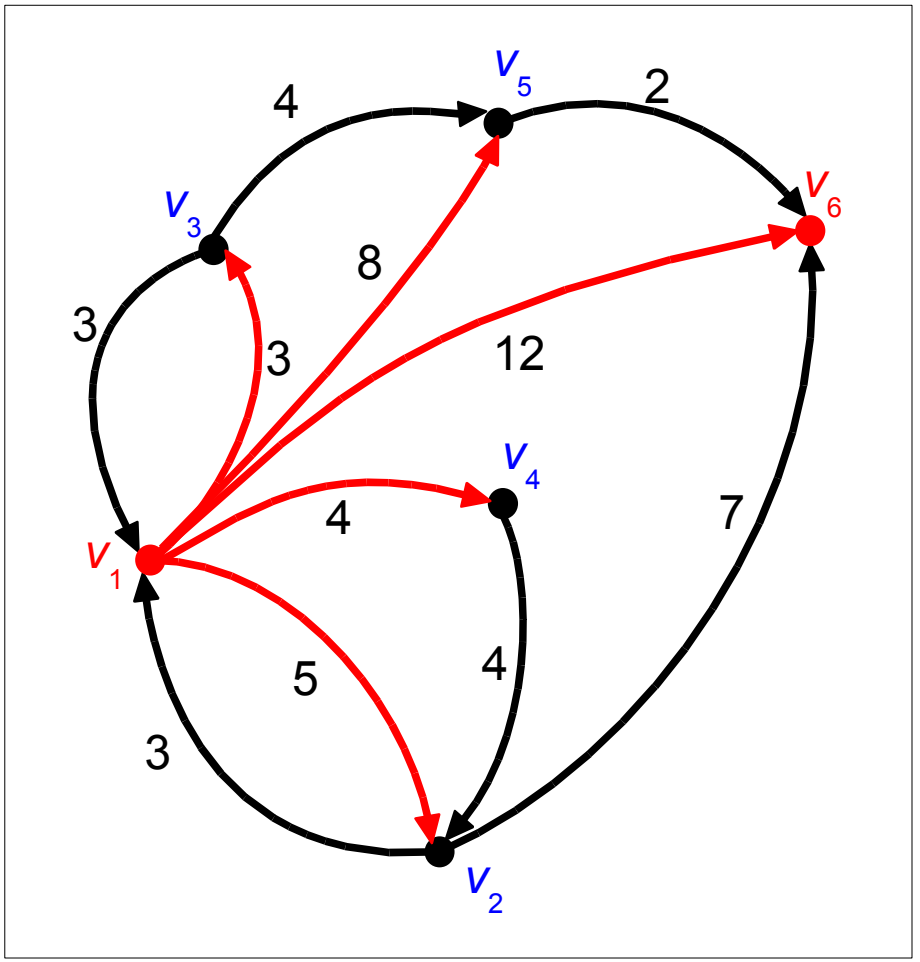
$Q = \{v_1\}$



v_1	v_2	v_3	v_4	v_5	v_6
0	∞	∞	∞	∞	∞

Algorytm BELLMANA-FORDA

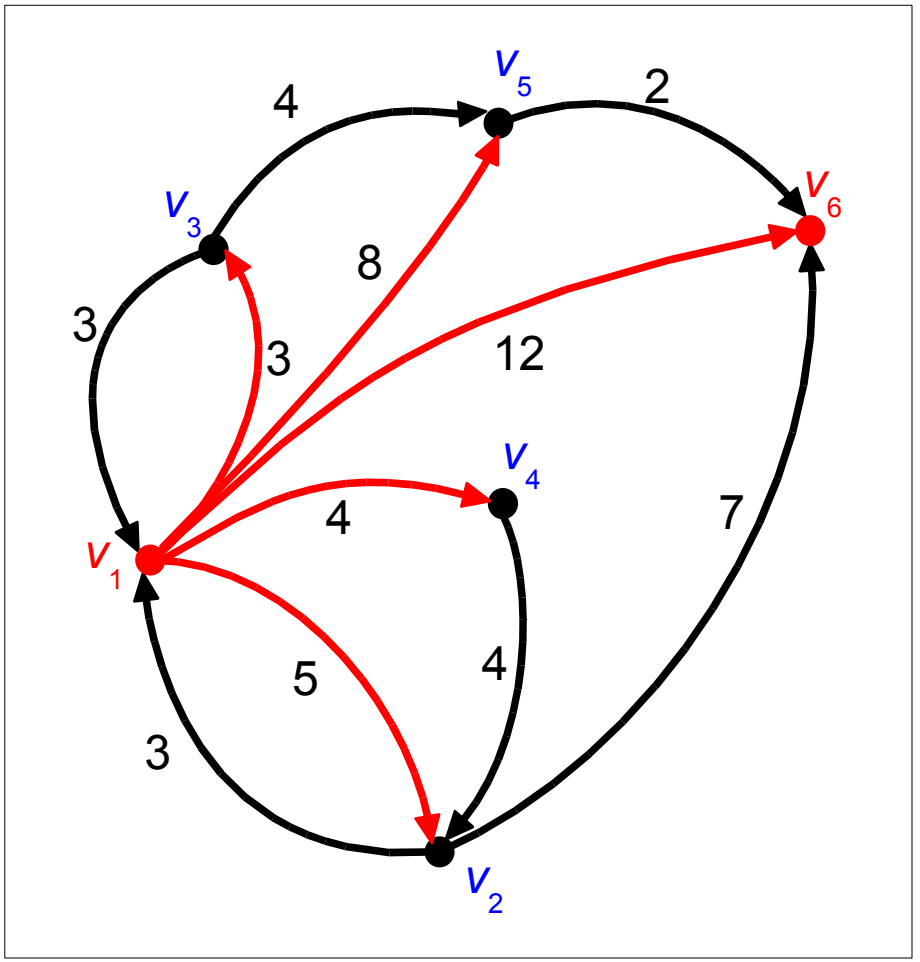
$Q = \{\ \}$



v_1	v_2	v_3	v_4	v_5	v_6
0	∞	∞	∞	∞	∞

Algorytm BELLMANA-FORDA

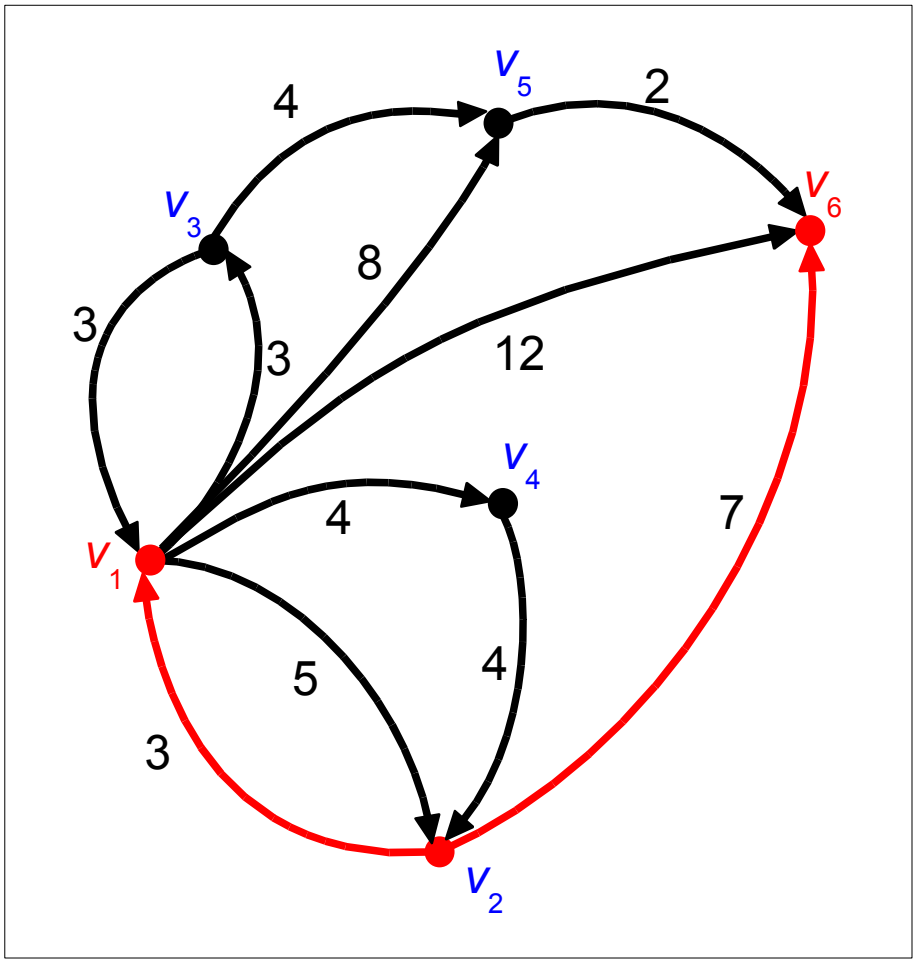
$Q = \{v_2, v_3, v_4, v_5, v_6\}$



v_1	v_2	v_3	v_4	v_5	v_6
0	5	3	4	8	12

Algorytm BELLMANA-FORDA

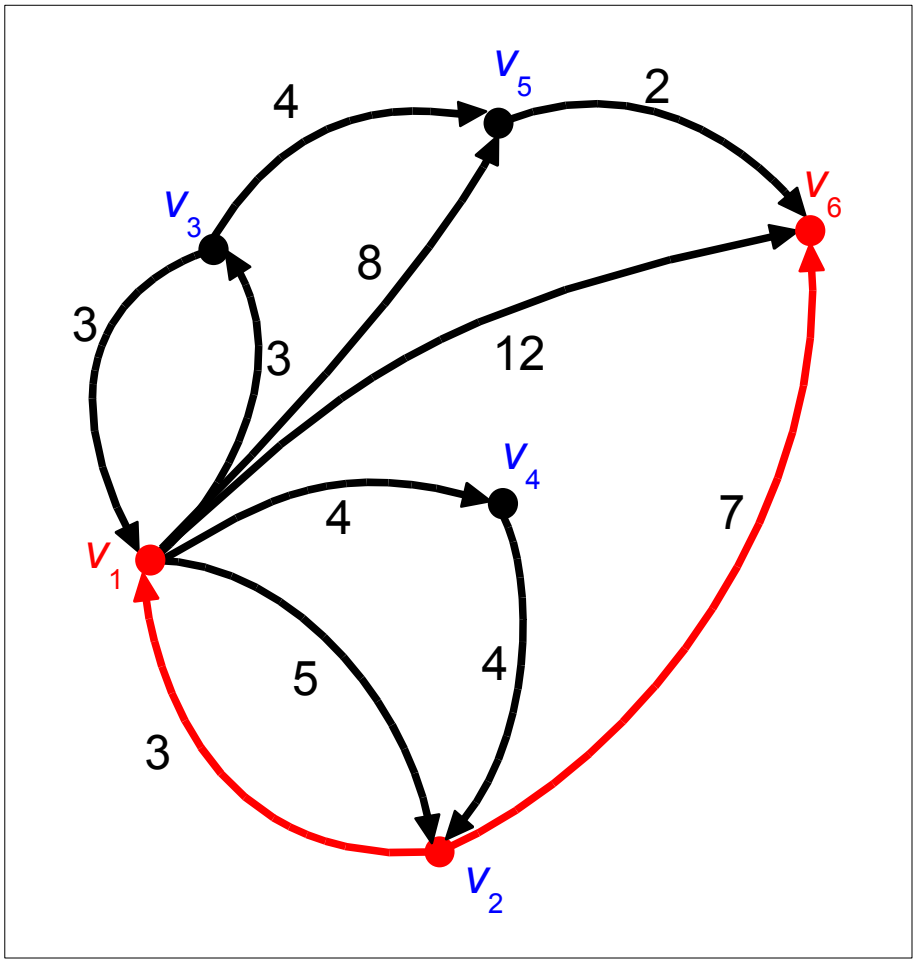
$Q = \{v_3, v_4, v_5, v_6\}$



v_1	v_2	v_3	v_4	v_5	v_6
0	5	3	4	8	12

Algorytm BELLMANA-FORDA

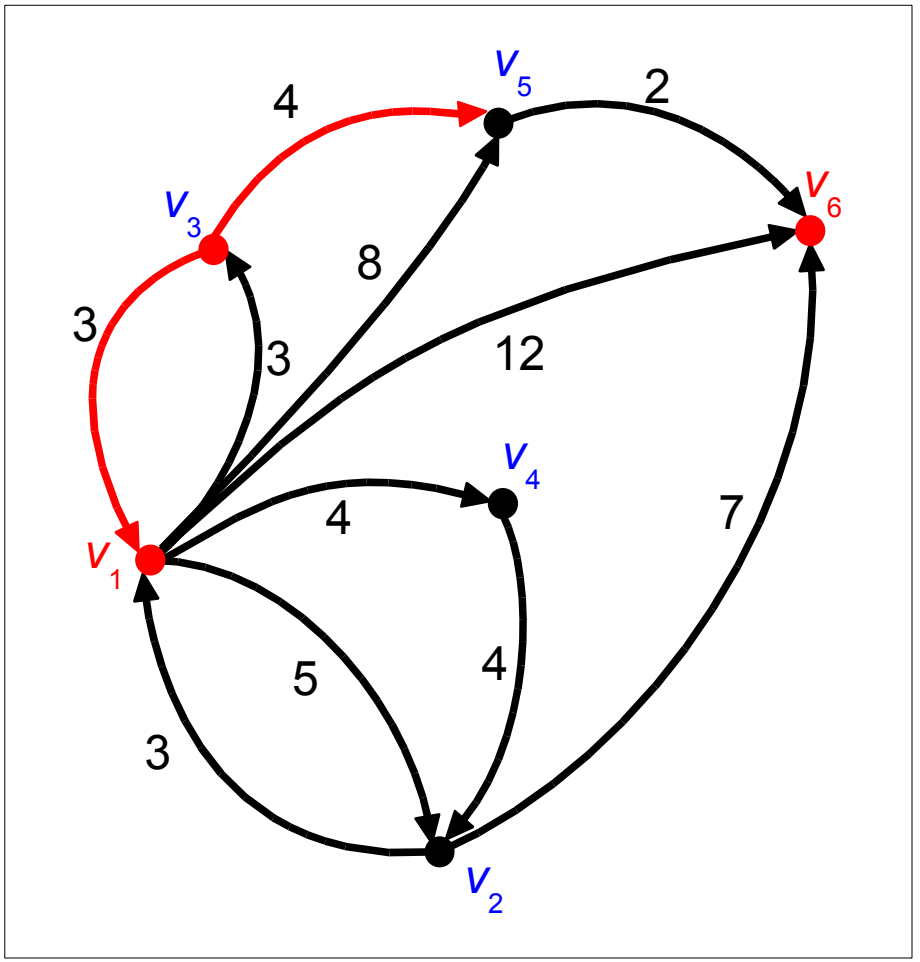
$Q = \{v_3, v_4, v_5, v_6\}$



v_1	v_2	v_3	v_4	v_5	v_6
0	5	3	4	8	12

Algorytm BELLMANA-FORDA

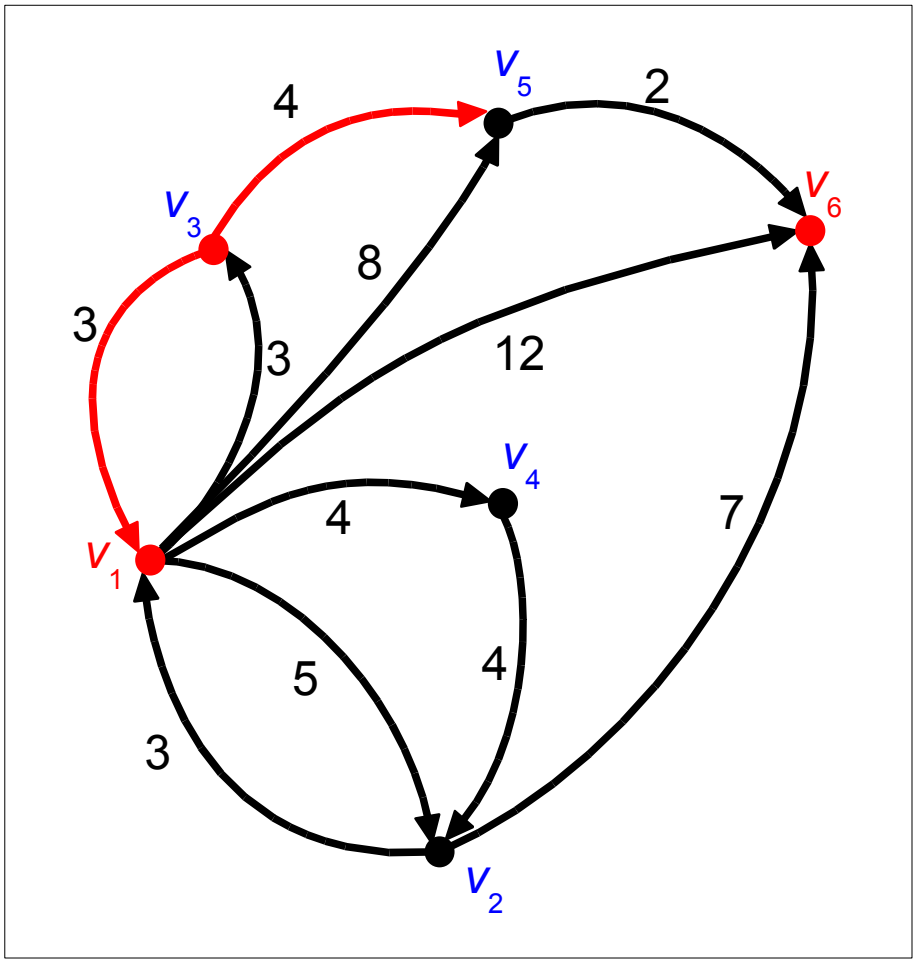
$Q = \{v_4, v_5, v_6\}$



v_1	v_2	v_3	v_4	v_5	v_6
0	5	3	4	8	12

Algorytm BELLMANA-FORDA

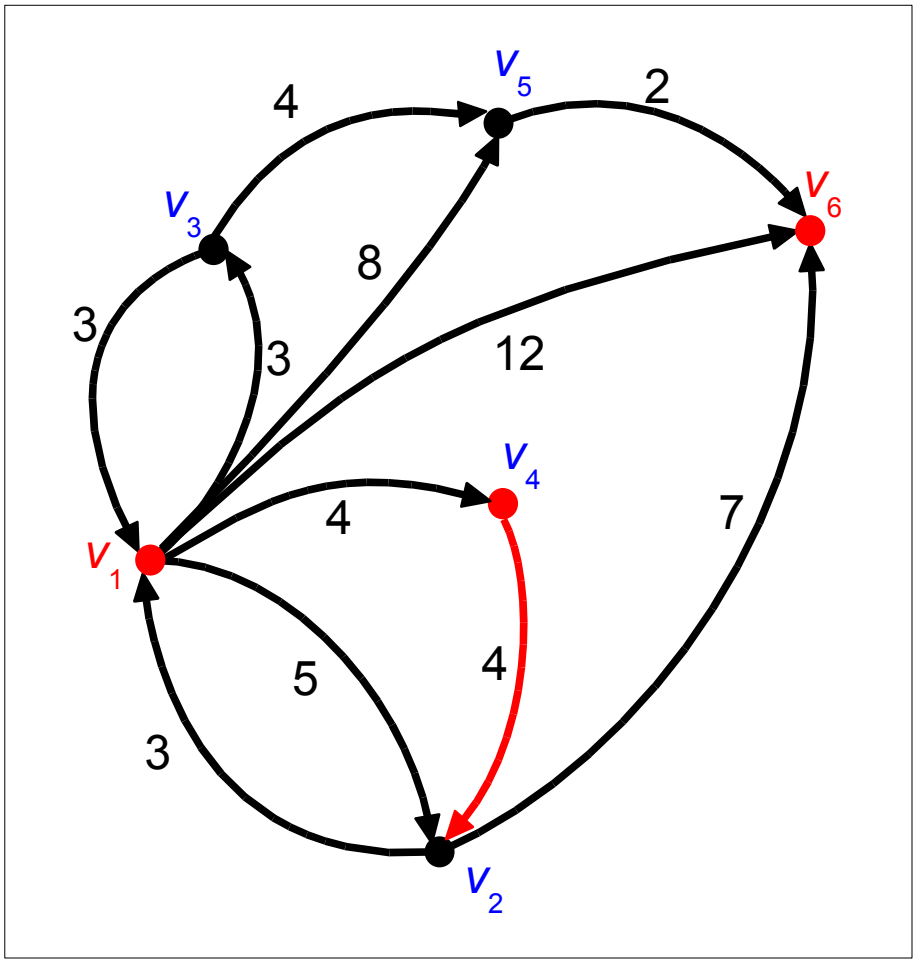
$Q = \{v_4, v_5, v_6\}$



v_1	v_2	v_3	v_4	v_5	v_6
0	5	3	4	7	12

Algorytm BELLMANA-FORDA

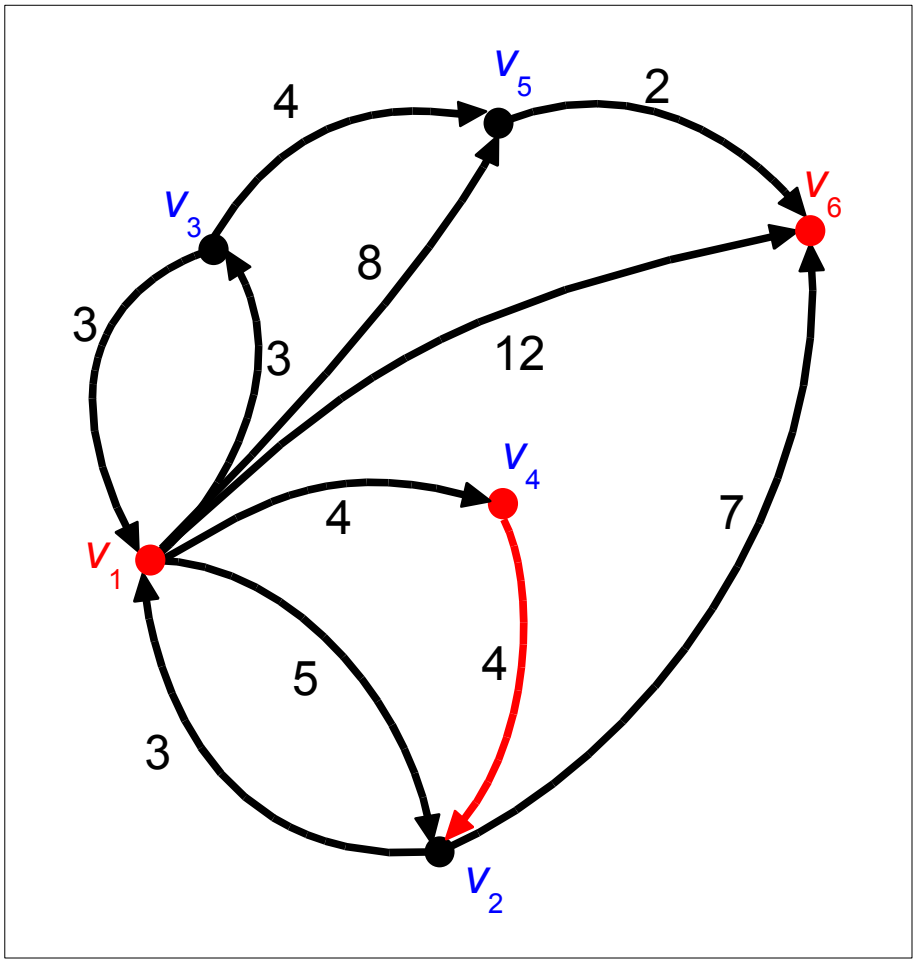
$Q = \{v_5, v_6\}$



v_1	v_2	v_3	v_4	v_5	v_6
0	5	3	4	7	12

Algorytm BELLMANA-FORDA

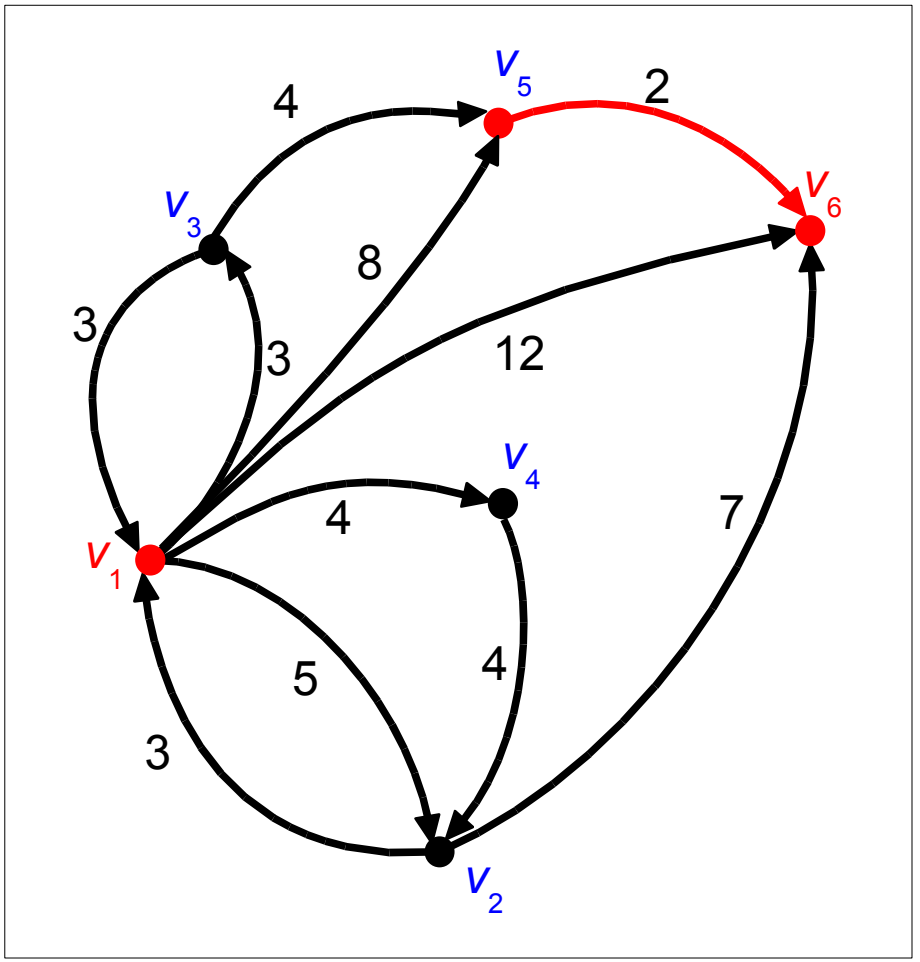
$Q = \{v_5, v_6\}$



v_1	v_2	v_3	v_4	v_5	v_6
0	5	3	4	7	12

Algorytm BELLMANA-FORDA

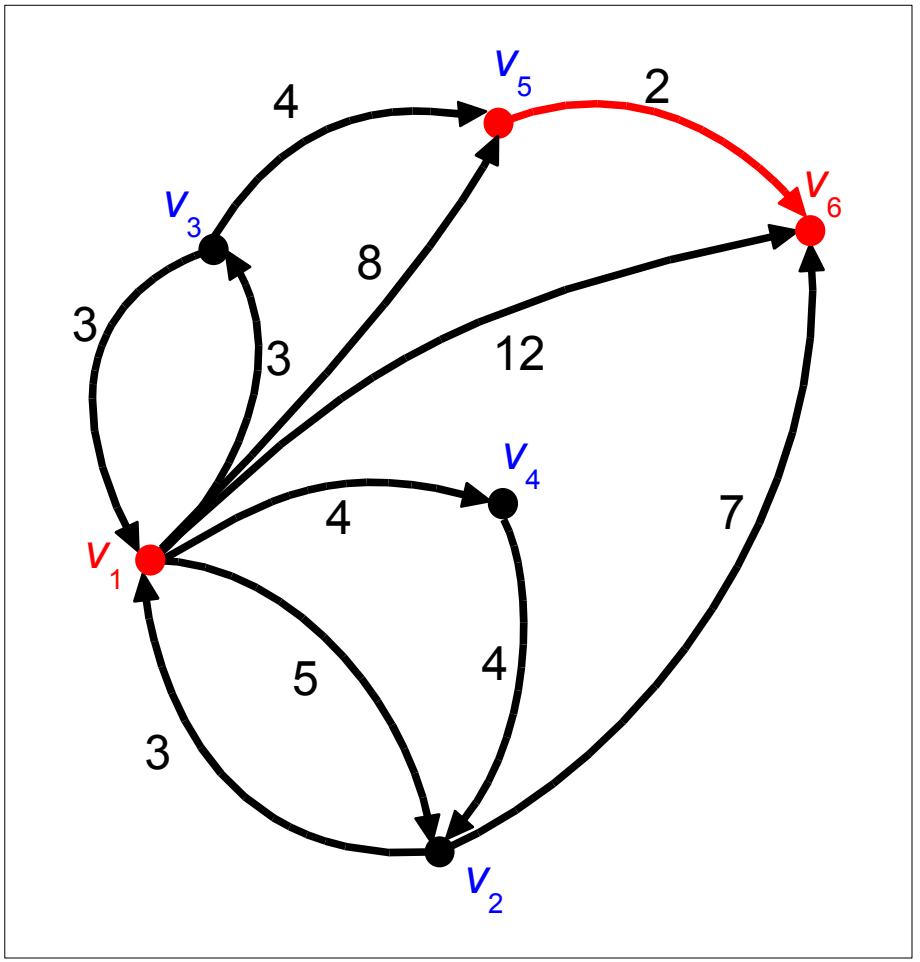
$Q = \{v_6\}$



v_1	v_2	v_3	v_4	v_5	v_6
0	5	3	4	7	12

Algorytm BELLMANA-FORDA

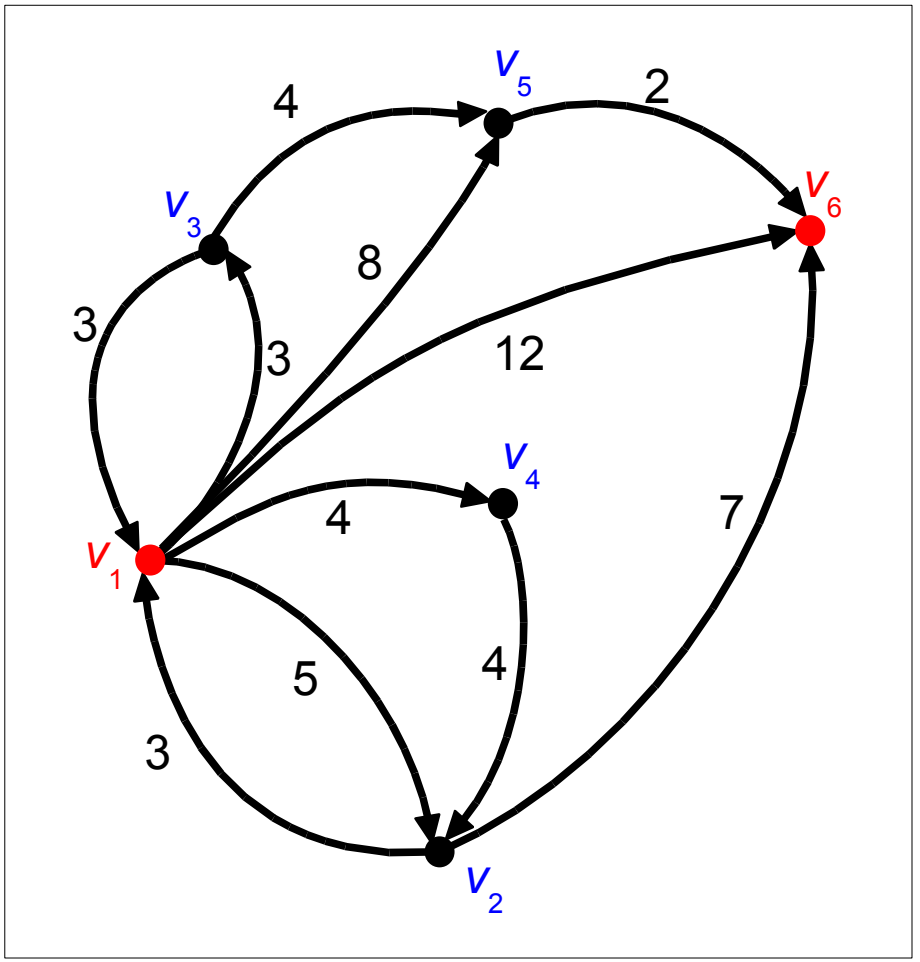
$Q = \{v_6\}$



v_1	v_2	v_3	v_4	v_5	v_6
0	5	3	4	7	9

Algorytm BELLMANA-FORDA

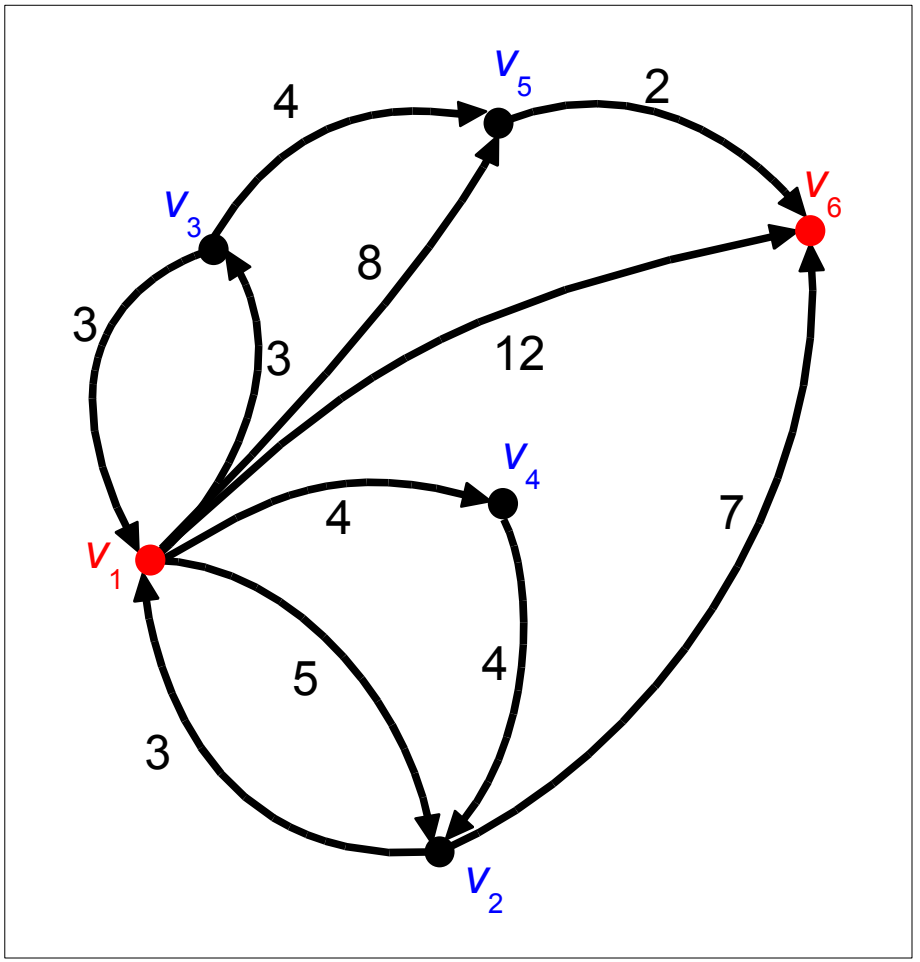
$Q = \{\ \}$



v_1	v_2	v_3	v_4	v_5	v_6
0	5	3	4	7	9

Algorytm BELLMANA-FORDA

$Q = \{\ \}$



v_1	v_2	v_3	v_4	v_5	v_6
0	5	3	4	7	**9**

Wyznaczenie **złożoności obliczeniowej** powyższej implementacji jest kłopotliwe (węzły mogą trafiać wielokrotnie do kolejki) – można wyznaczyć nawet takie przypadki, dla których będzie to złożoność wykładnicza. Jednak zazwyczaj jest to mało realne.

Oczywiście **optymalność** algorytmu jest zachowana.

Rozpatrzmy jeszcze **szczególny przypadek** zagadnienia:

najkrótsza droga ze źródła do określonego (pojedynczego) węzła.

W takim wypadku (i przy nieujemnych wagach) przewagę ma algorytm Dijkstry, gdyż możemy go **zakończyć** nie po n iteracjach, ale gdy zostanie **ustalone oszacowanie węzła końcowego** (przeznaczenia).

W najgorszym wypadku może to być oczywiście ostatnia (n-ta) iteracja, ale często będzie to któraś z iteracji wcześniejszych.

Wyznaczanie maksymalnego przepływu w sieciach

W tym zagadnieniu wagi łuków nie reprezentują odległości (jak przy najkrótszych drogach), tylko maksymalny możliwy przepływ pomiędzy węzłami (informacji, płynu, gazu, itp.). Problem polega na wyznaczeniu takich wartości przepływu pomiędzy węzłami (nie większych niż maksymalne), aby przepływ ze źródła do określonego węzła końcowego był jak największy.

Algorytm FORDA-FULKERSONA

dotyczy acyklicznych grafów skierowanych. Jest to jedna z podstawowych idei w tym zakresie.

Najogólniej, polega ona na **sukcesywnym wyszukiwaniu (dowolnych) dróg ze źródła do ujścia** (węzła końcowego).

Można to zrobić w ten sposób, że będąc w źródle, wybieramy pierwszy węzeł incydentny (należy ustalić kolejność węzłów), z niego znowu wybieramy pierwszy incydentny, itd.

W efekcie, albo dotrzemy do ujścia, albo do innego węzła końcowego (z którego nie ma żadnych łuków wychodzących):

- **Jeśli dotarliśmy do węzła nie będącego ujściem**, to **cofamy** się do węzła poprzedzającego i wybieramy kolejny węzeł incydentny (jeśli trzeba, to cofamy się o więcej niż 1 węzeł).
  Łuki do węzłów, z których się wycofujemy (gdy nie można z nich osiągnąć ujścia), usuwamy.
- **Jeśli ostatni węzeł jest ujściem**, to **mamy wyznaczoną drogę**.

Każdorazowo, **po wyznaczeniu jakiejś drogi**, znajdowany jest w niej łuk o najmniejszym przepływie i daną drogą **ustalamy przepływ** o takiej właśnie wartości, tzn. od przepływu każdego łuku z danej drogi odejmujemy tą najmniejszą wartość.

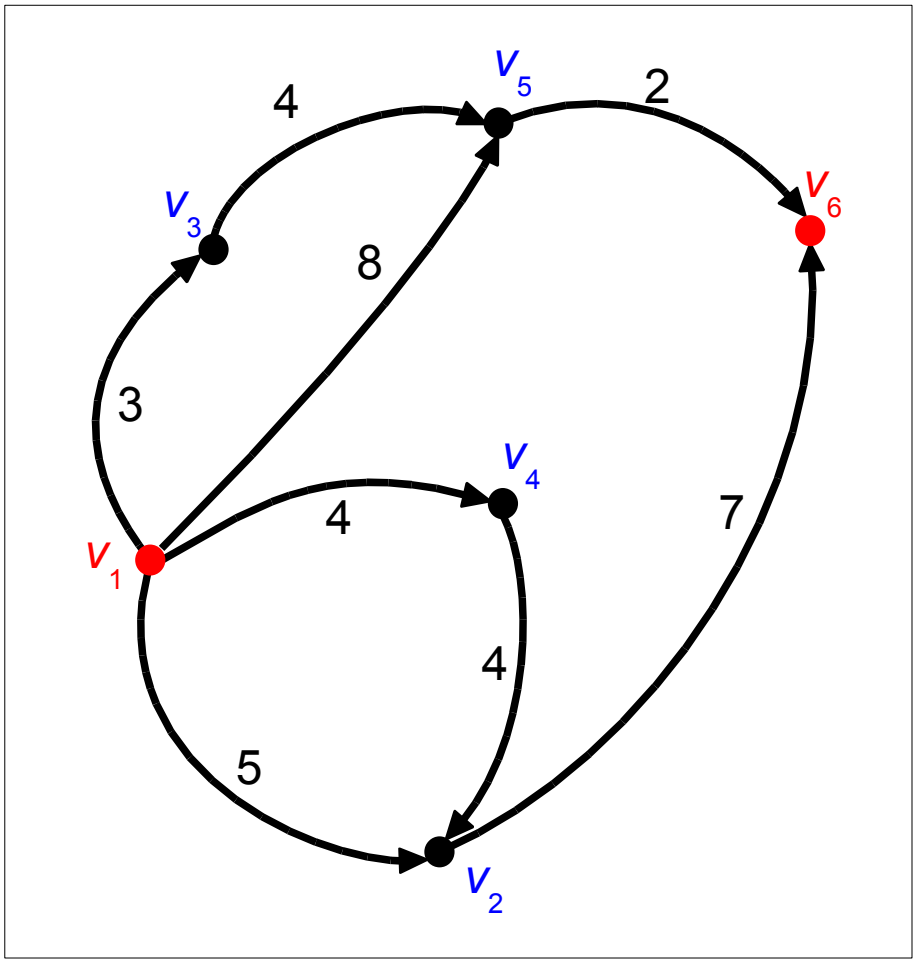
Następnie łuki o zerowym przepływie usuwamy z grafu.

Algorytm **kończy działanie**, gdy usuniemy wszystkie łuki wychodzące ze źródła.

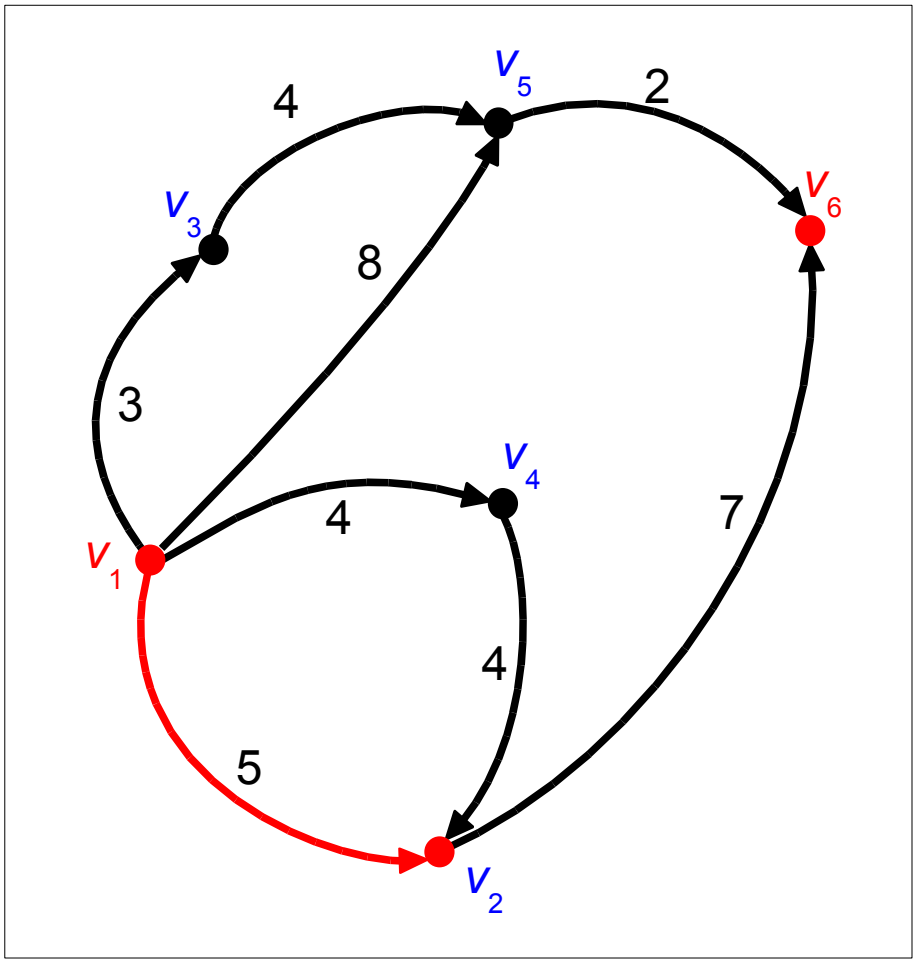
Wartość przepływu maksymalnego uzyskujemy poprzez zsumowanie przepływów ze wszystkich wyznaczonych dróg.

Wykazano, że **czas działania** algorytmu może wzrosnąć do nieskończoności, jeśli wybór kolejnych węzłów w drogach będzie dowolny.
Jeśli natomiast zawsze będziemy wyszukiwali drogi najkrótsze (o najmniejszej liczbie łuków), to uzyskamy algorytm wielomianowy $O(nm^2)$. Taką wersję algorytmu nazywamy **algorytmem Edmondsa-Karpa (1969)**.

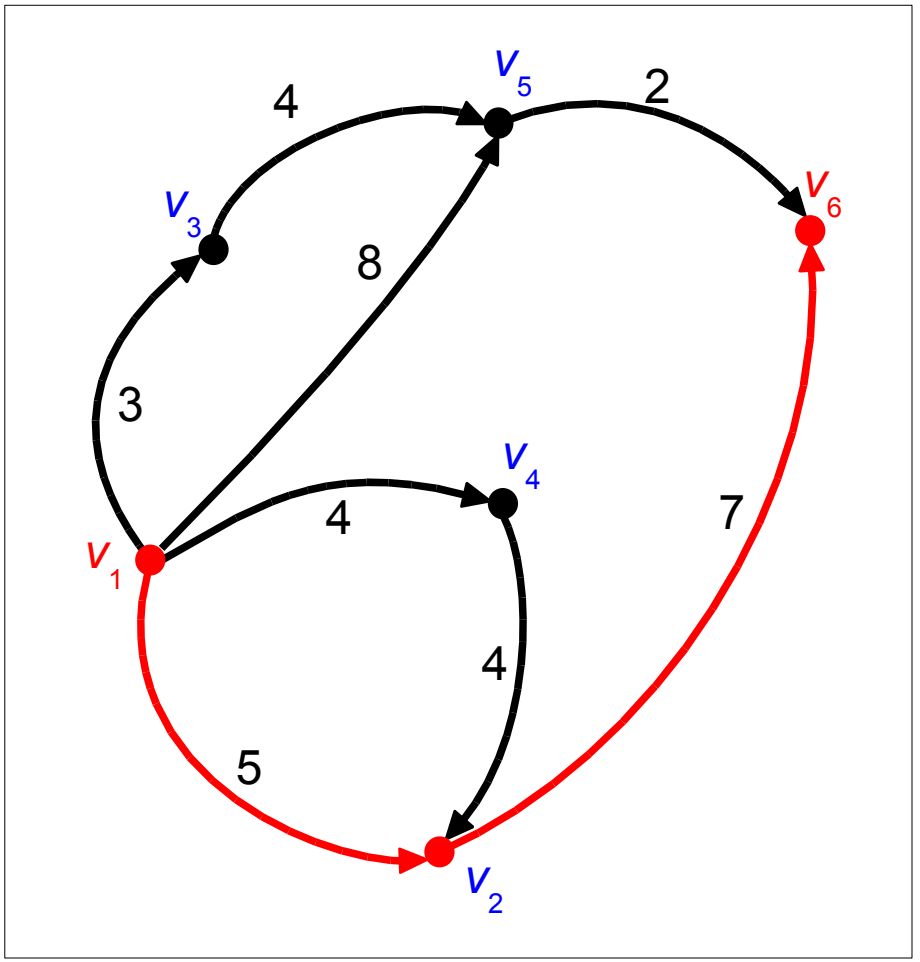
Algorytm FORDA-FULKERSONA



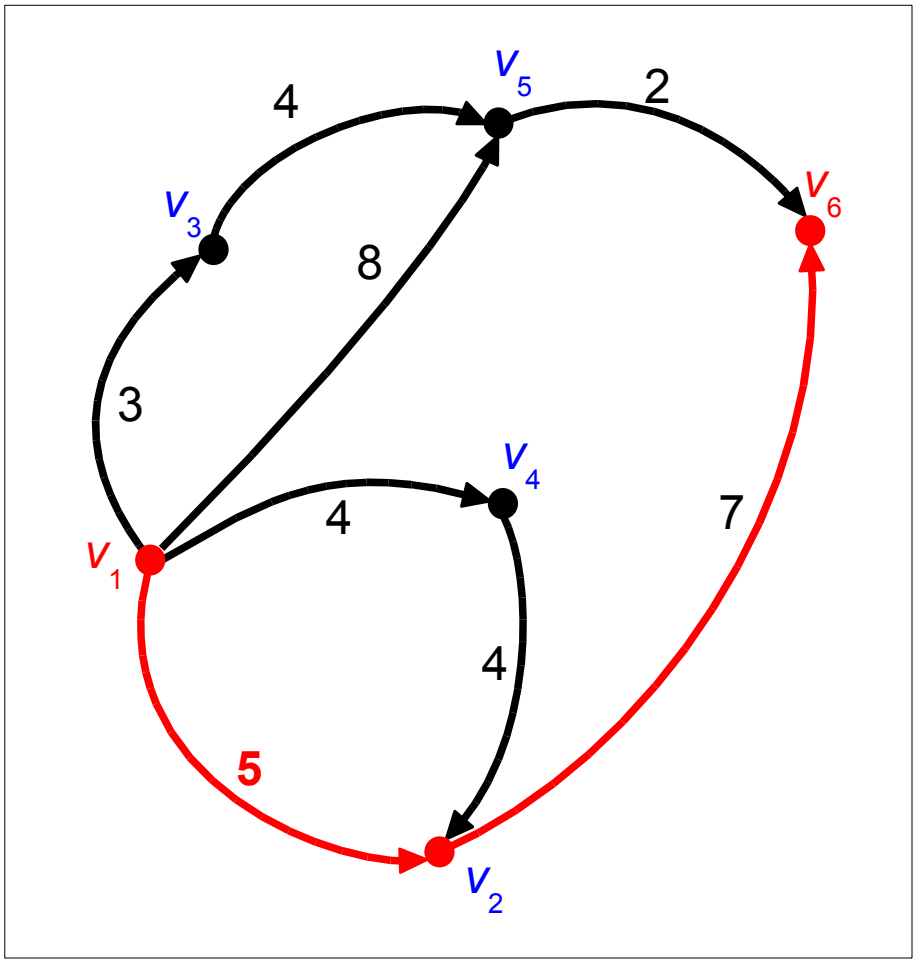
Algorytm FORDA-FULKERSONA



Algorytm FORDA-FULKERSONA



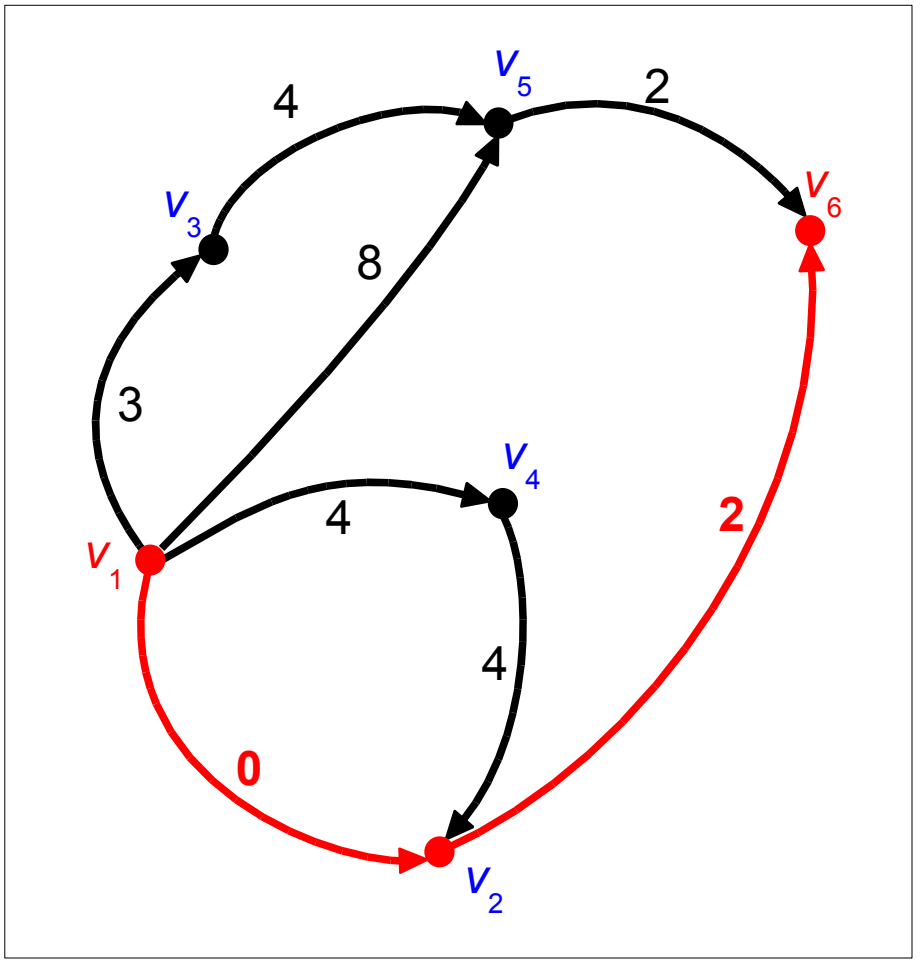
Algorytm FORDA-FULKERSONA



Przepływy:

$F(v_1,v_2,v_6) = \mathbf{5}$

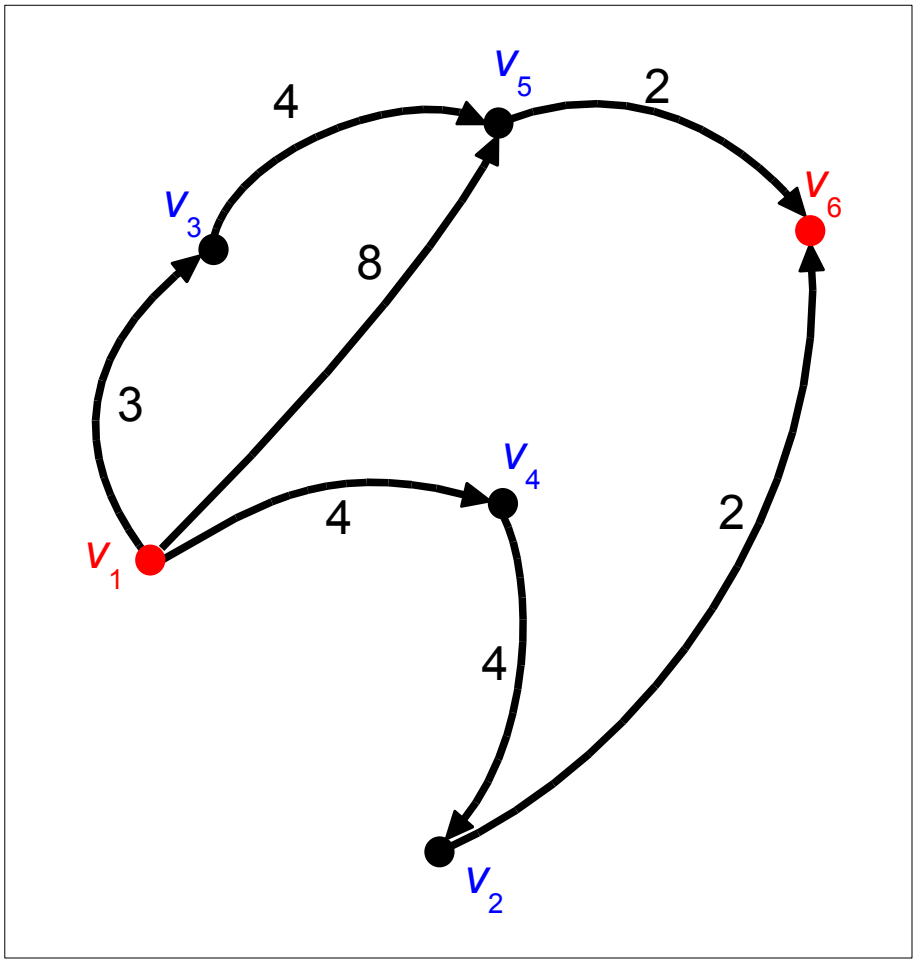
Algorytm FORDA-FULKERSONA



Przepływy:

$F(v_1, v_2, v_6) = \mathbf{5}$

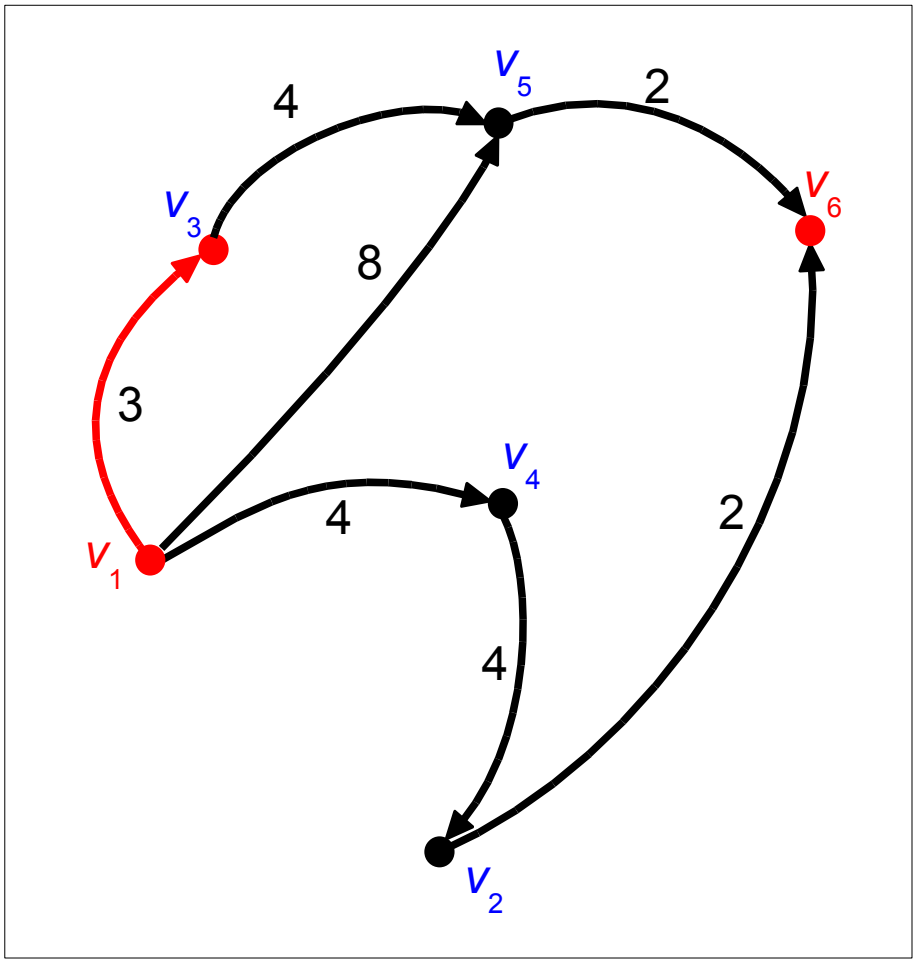
Algorytm FORDA-FULKERSONA



Przepływy:

$F(v_1,v_2,v_6) = \mathbf{5}$

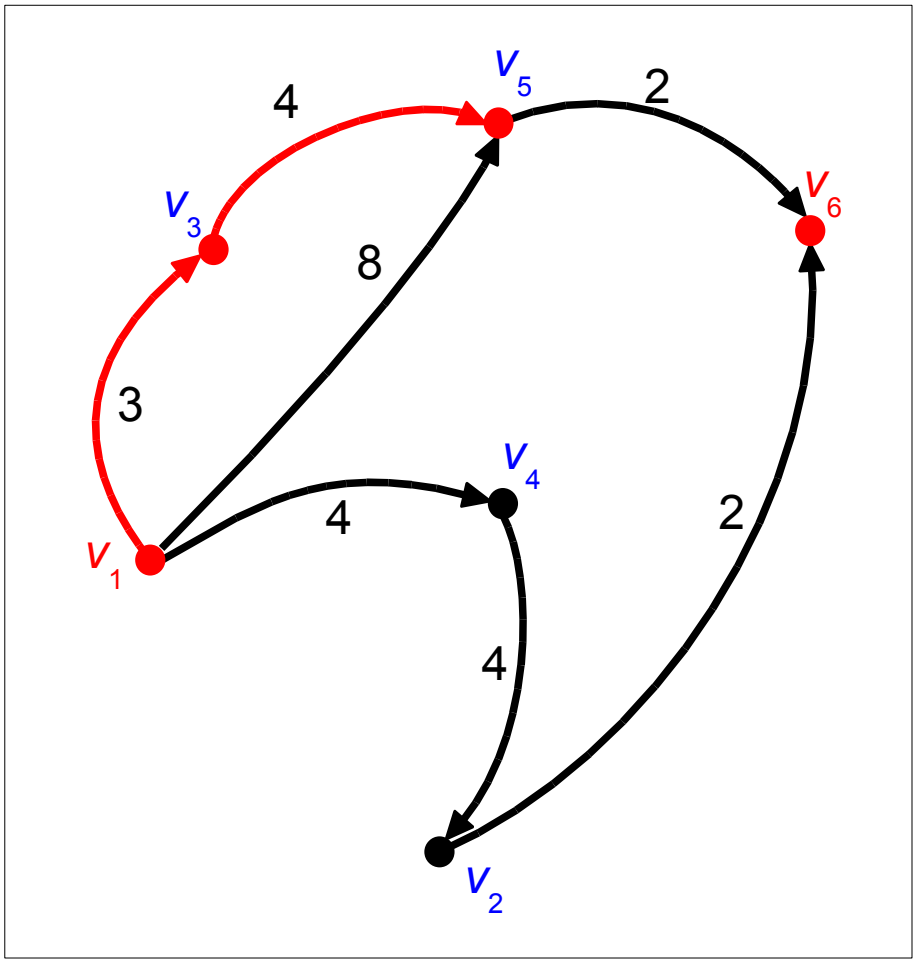
Algorytm FORDA-FULKERSONA



Przepływy:

$F(v_1,v_2,v_6) = \mathbf{5}$

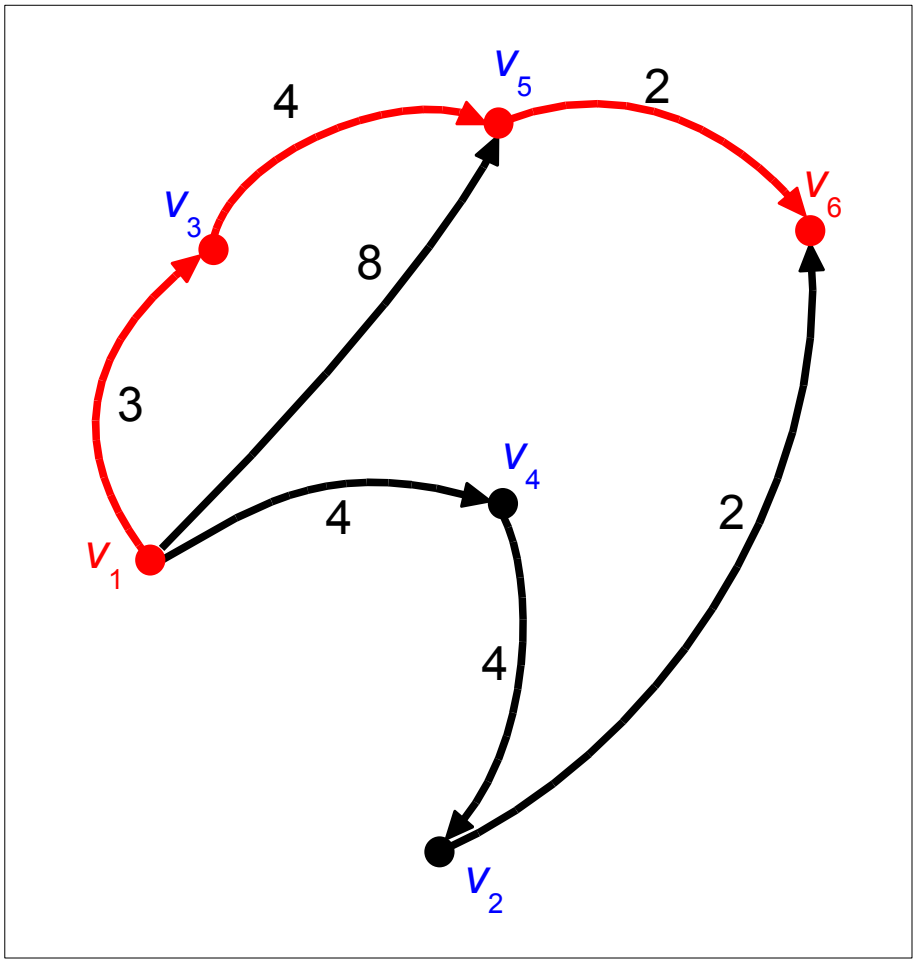
Algorytm FORDA-FULKERSONA



Przepływy:

$F(v_1,v_2,v_6) = \mathbf{5}$

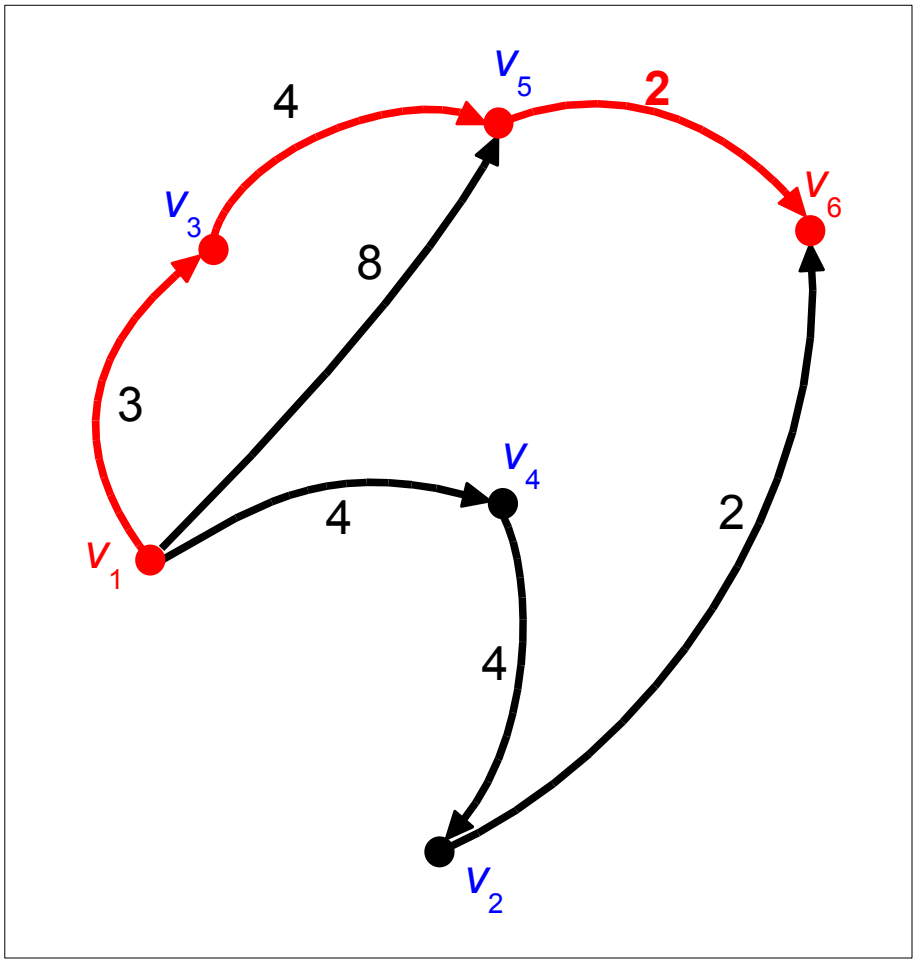
Algorytm FORDA-FULKERSONA



Przepływy:

$F(v_1,v_2,v_6) = \mathbf{5}$

Algorytm FORDA-FULKERSONA

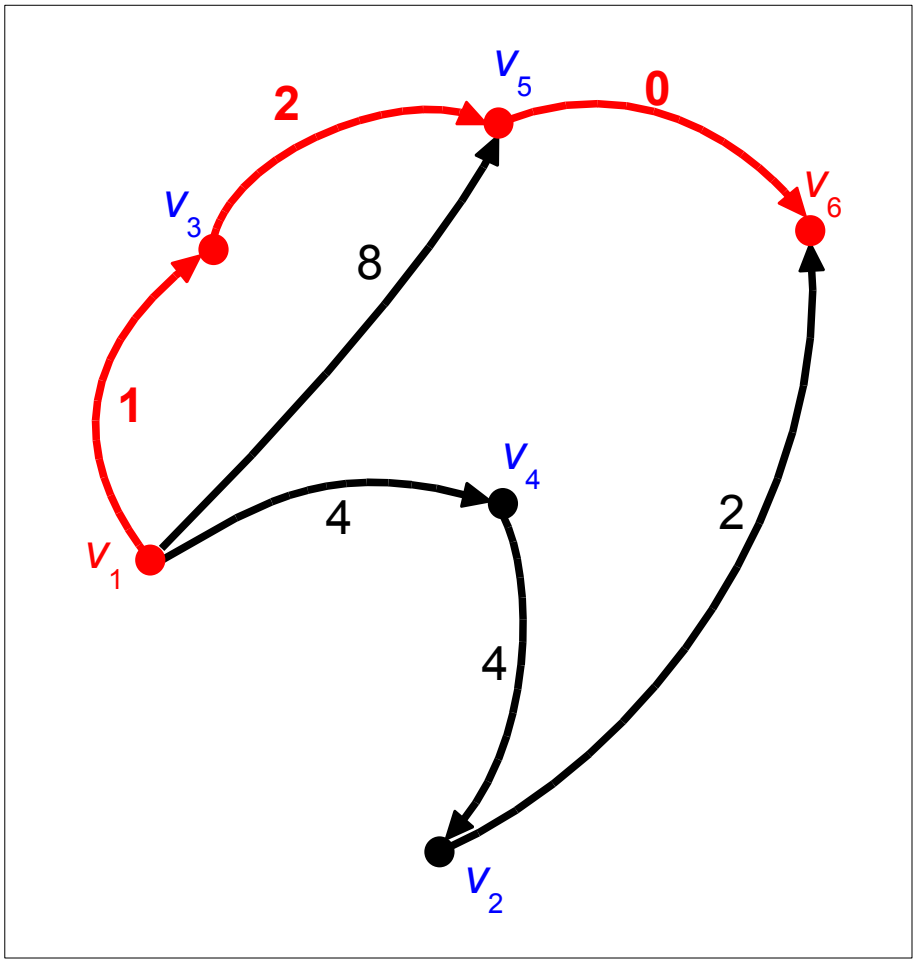


Przepływy:

$F(v_1, v_2, v_6) = \mathbf{5}$

$F(v_1, v_3, v_5, v_6) = \mathbf{2}$

Algorytm FORDA-FULKERSONA

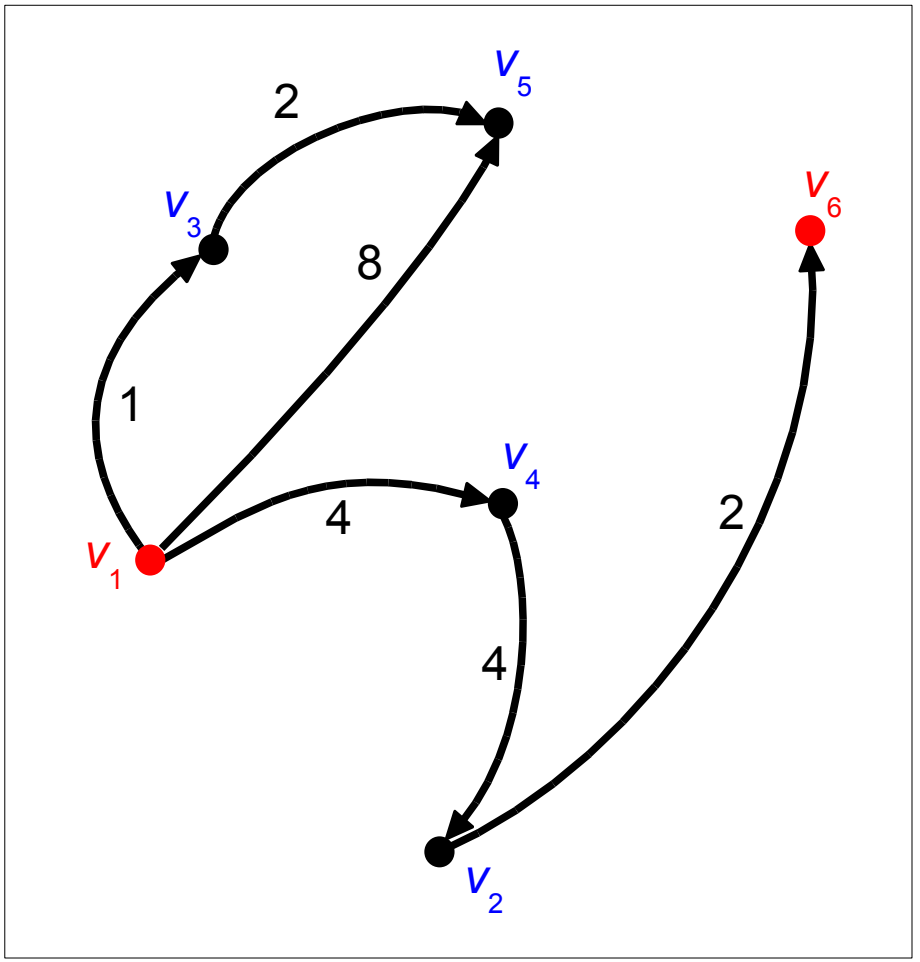


Przepływy:

$F(v_1, v_2, v_6) = \mathbf{5}$

$F(v_1, v_3, v_5, v_6) = \mathbf{2}$

Algorytm FORDA-FULKERSONA

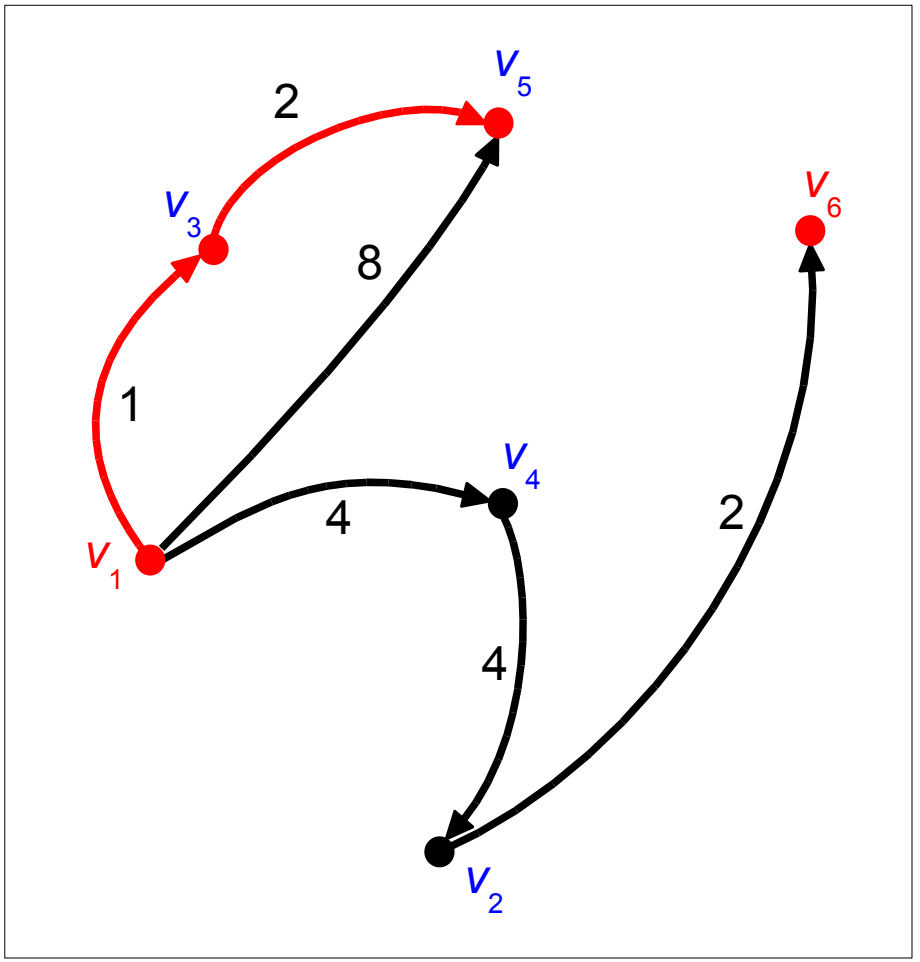


Przepływy:

$F(v_1, v_2, v_6) = \mathbf{5}$

$F(v_1, v_3, v_5, v_6) = \mathbf{2}$

Algorytm FORDA-FULKERSONA

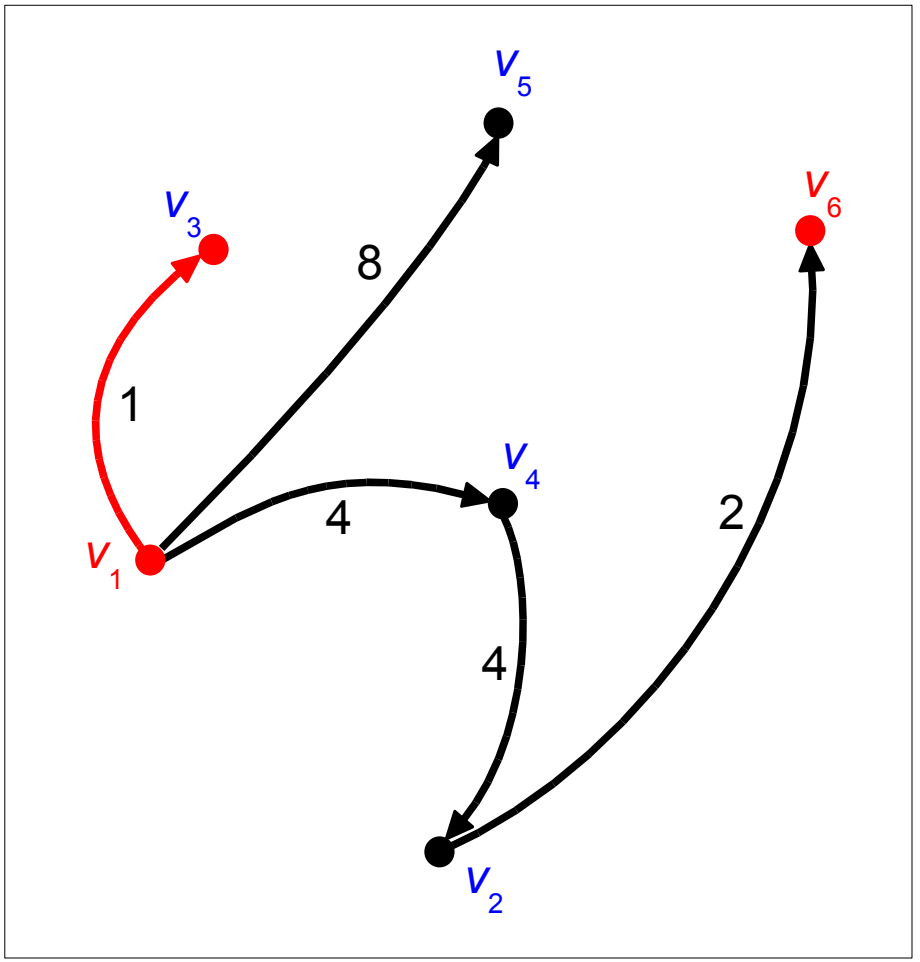


Przepływy:

$F(v_1, v_2, v_6) = \mathbf{5}$

$F(v_1, v_3, v_5, v_6) = \mathbf{2}$

Algorytm FORDA-FULKERSONA

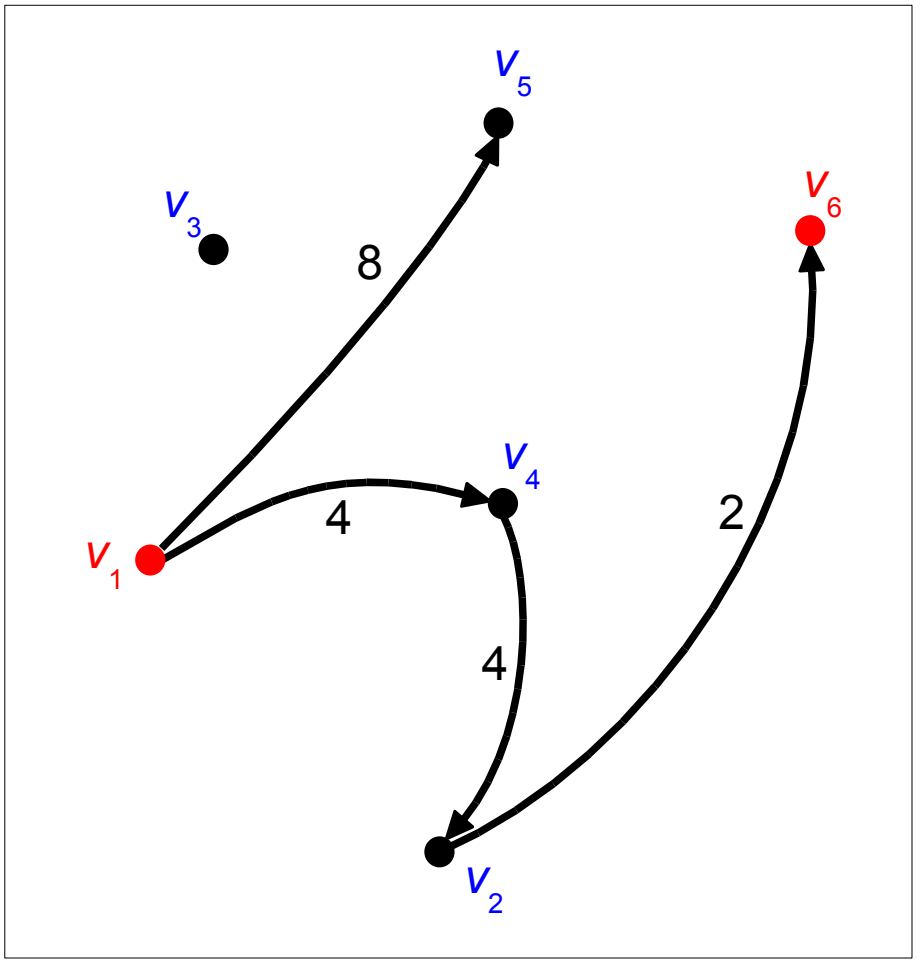


Przepływy:

$F(v_1, v_2, v_6) = \mathbf{5}$

$F(v_1, v_3, v_5, v_6) = \mathbf{2}$

Algorytm FORDA-FULKERSONA

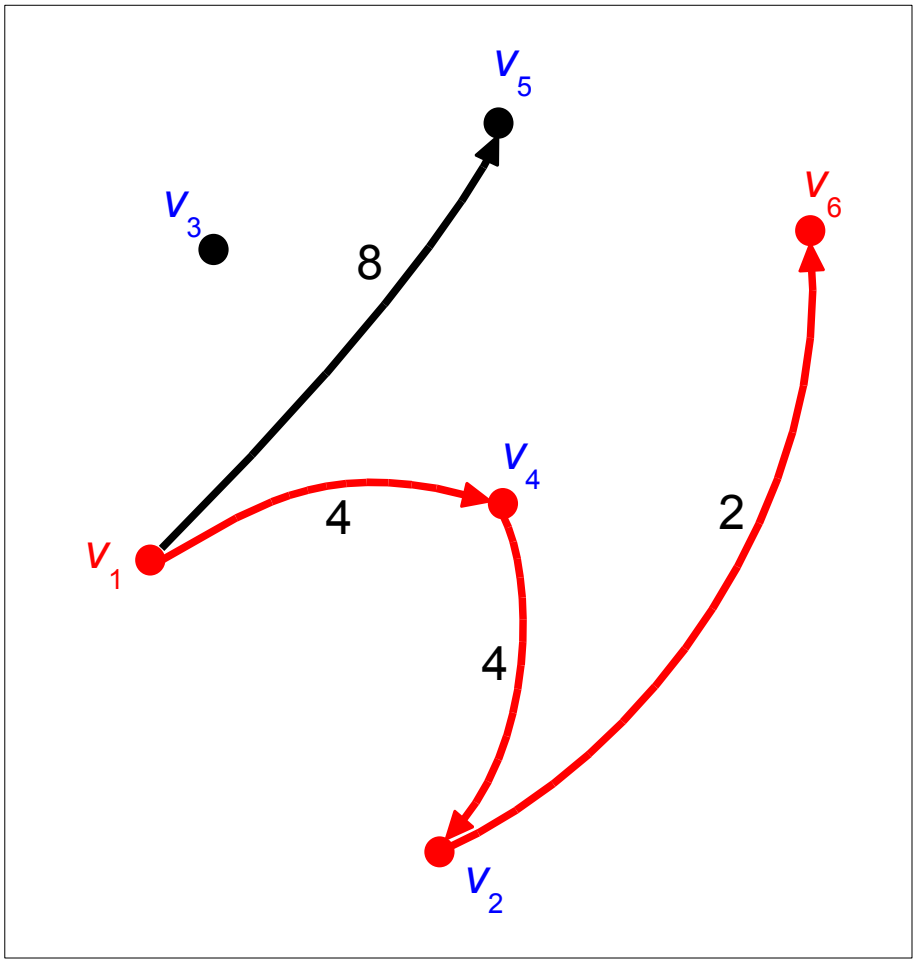


Przepływy:

$F(v_1, v_2, v_6) = \mathbf{5}$

$F(v_1, v_3, v_5, v_6) = \mathbf{2}$

Algorytm FORDA-FULKERSONA

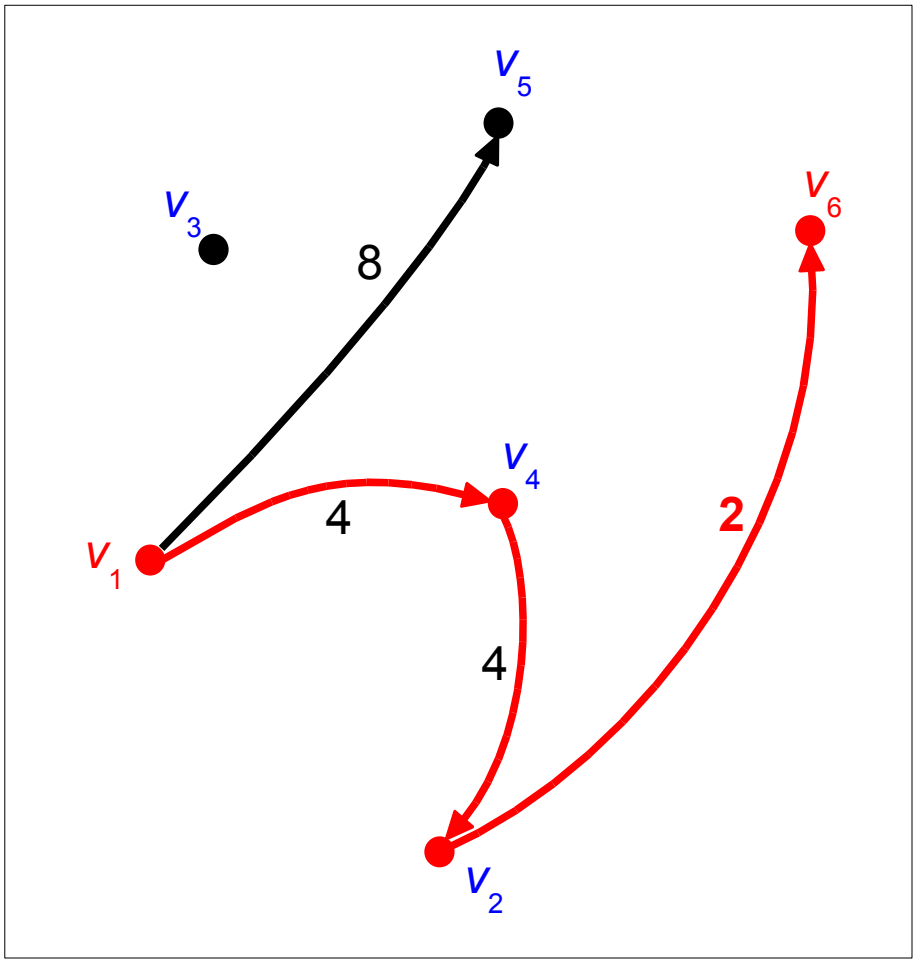


Przepływy:

$F(v_1, v_2, v_6) = \mathbf{5}$

$F(v_1, v_3, v_5, v_6) = \mathbf{2}$

Algorytm FORDA-FULKERSONA



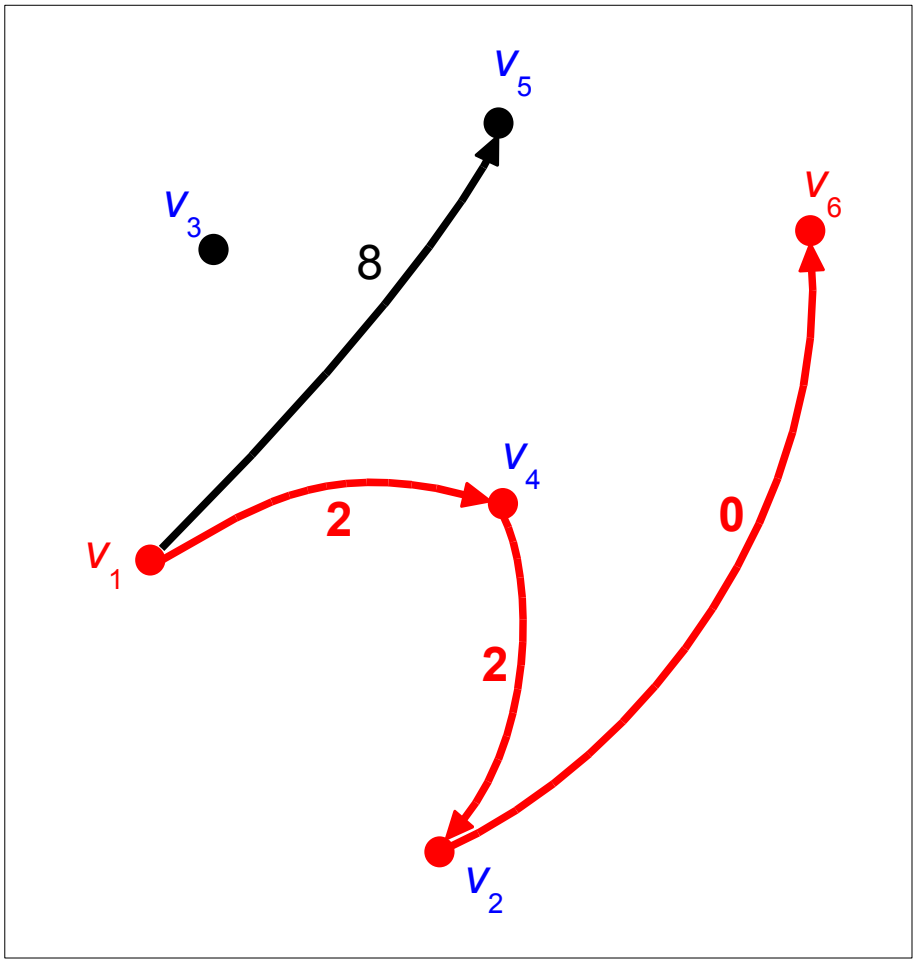
Przepływy:

$F(v_1, v_2, v_6) = \mathbf{5}$

$F(v_1, v_3, v_5, v_6) = \mathbf{2}$

$F(v_1, v_4, v_2, v_6) = \mathbf{2}$

Algorytm FORDA-FULKERSONA



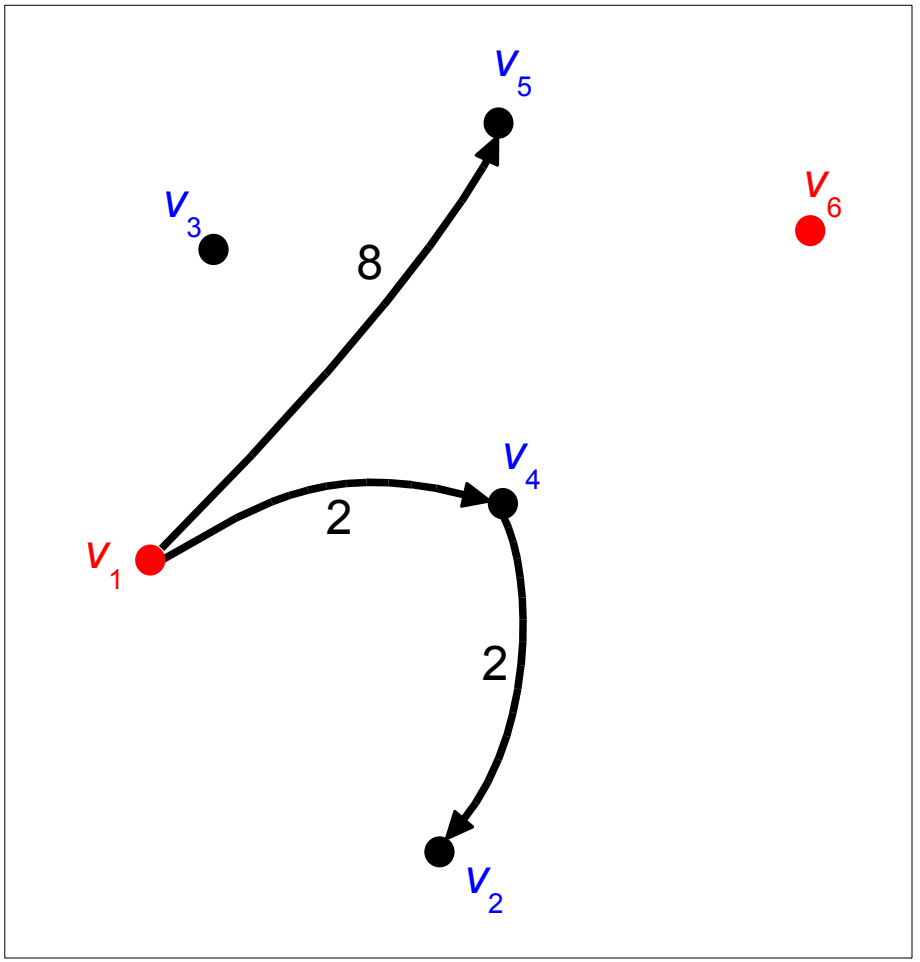
Przepływy:

$F(v_1, v_2, v_6) = \mathbf{5}$

$F(v_1, v_3, v_5, v_6) = \mathbf{2}$

$F(v_1, v_4, v_2, v_6) = \mathbf{2}$

Algorytm FORDA-FULKERSONA



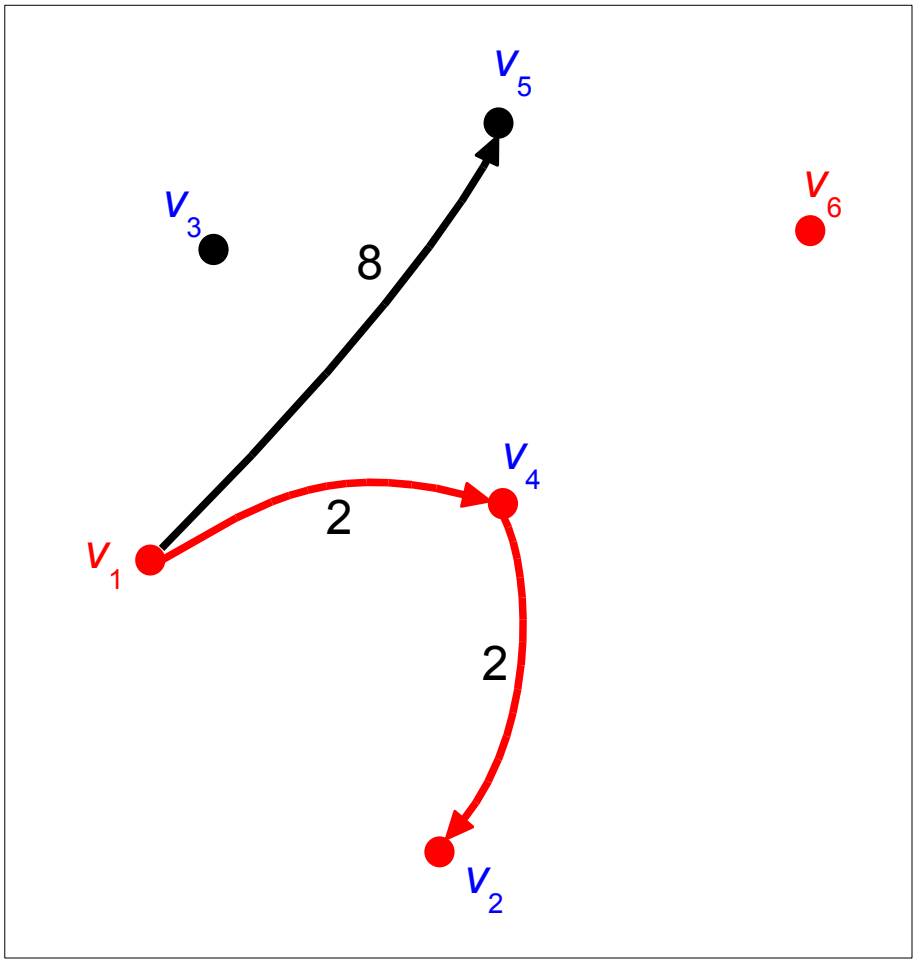
Przepływy:

$F(v_1, v_2, v_6) = \mathbf{5}$

$F(v_1, v_3, v_5, v_6) = \mathbf{2}$

$F(v_1, v_4, v_2, v_6) = \mathbf{2}$

Algorytm FORDA-FULKERSONA



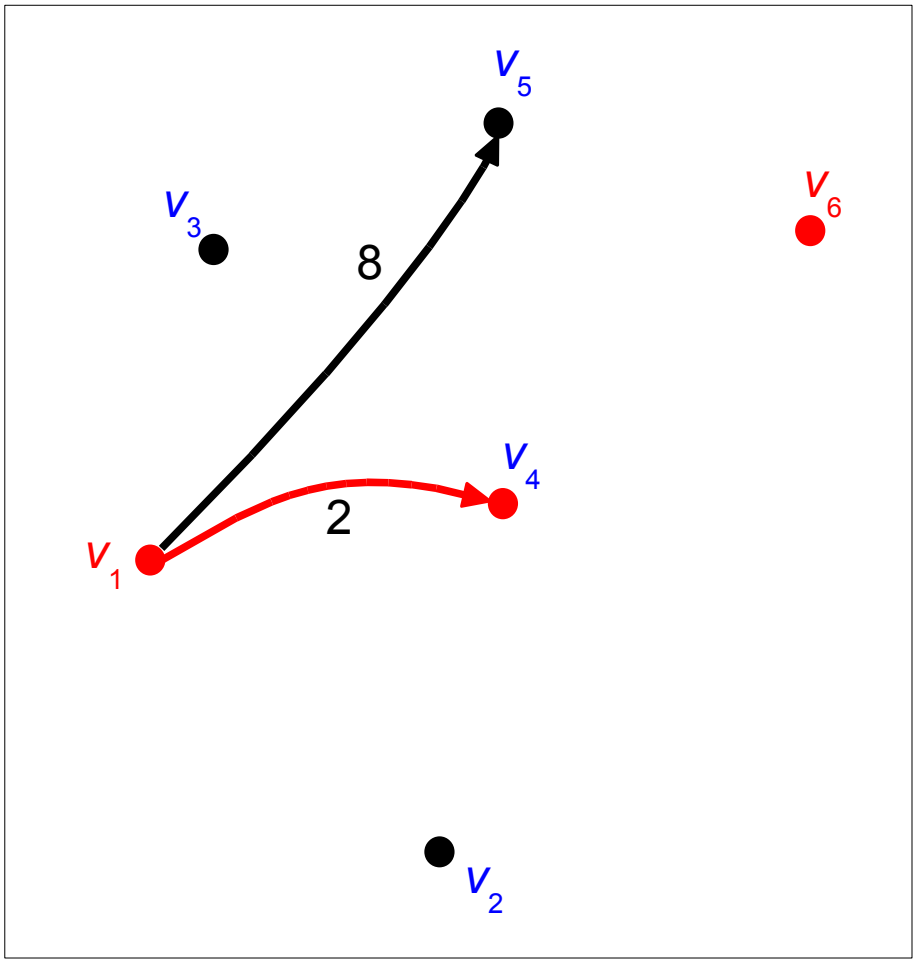
Przepływy:

$F(v_1, v_2, v_6) = \mathbf{5}$

$F(v_1, v_3, v_5, v_6) = \mathbf{2}$

$F(v_1, v_4, v_2, v_6) = \mathbf{2}$

Algorytm FORDA-FULKERSONA



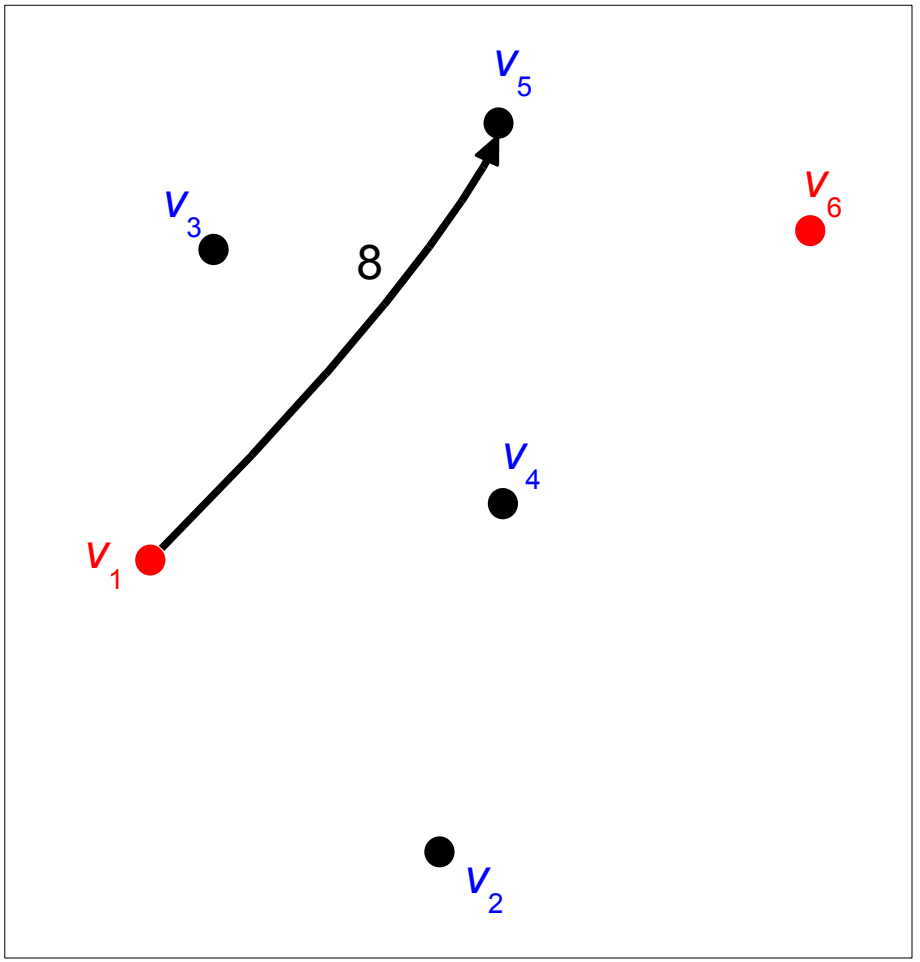
Przepływy:

$F(v_1, v_2, v_6) = \mathbf{5}$

$F(v_1, v_3, v_5, v_6) = \mathbf{2}$

$F(v_1, v_4, v_2, v_6) = \mathbf{2}$

Algorytm FORDA-FULKERSONA



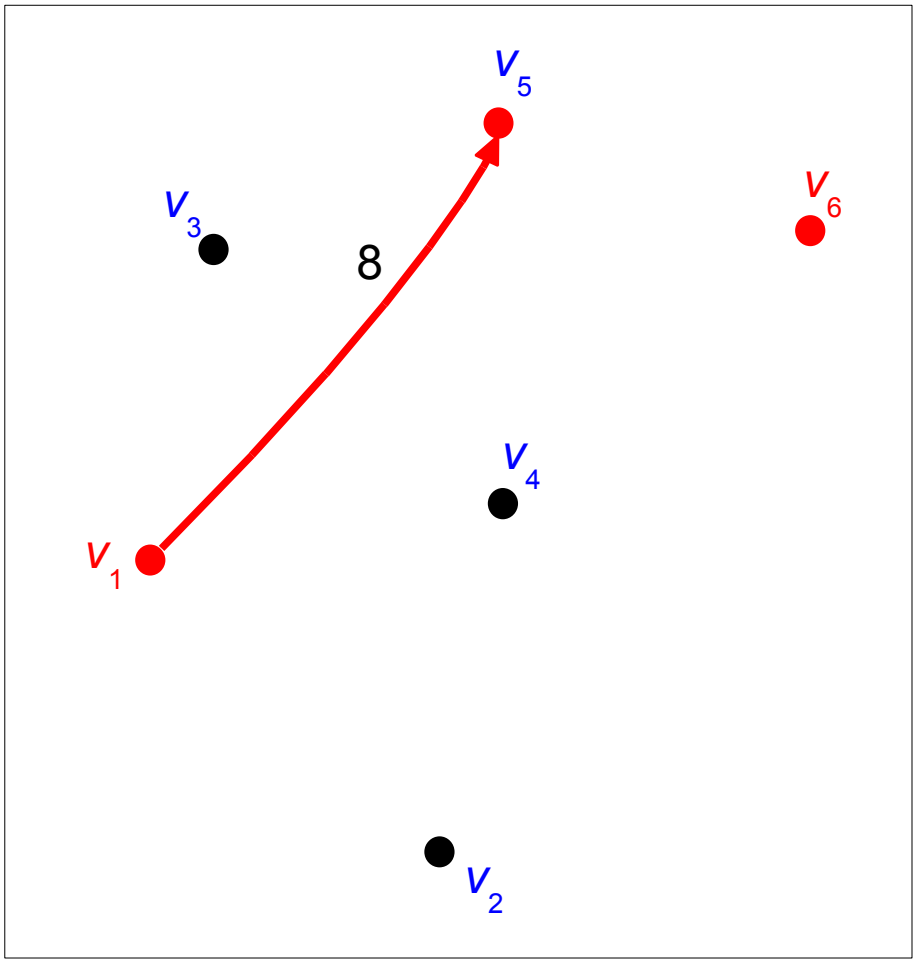
Przepływy:

$F(v_1, v_2, v_6) = \mathbf{5}$

$F(v_1, v_3, v_5, v_6) = \mathbf{2}$

$F(v_1, v_4, v_2, v_6) = \mathbf{2}$

Algorytm FORDA-FULKERSONA



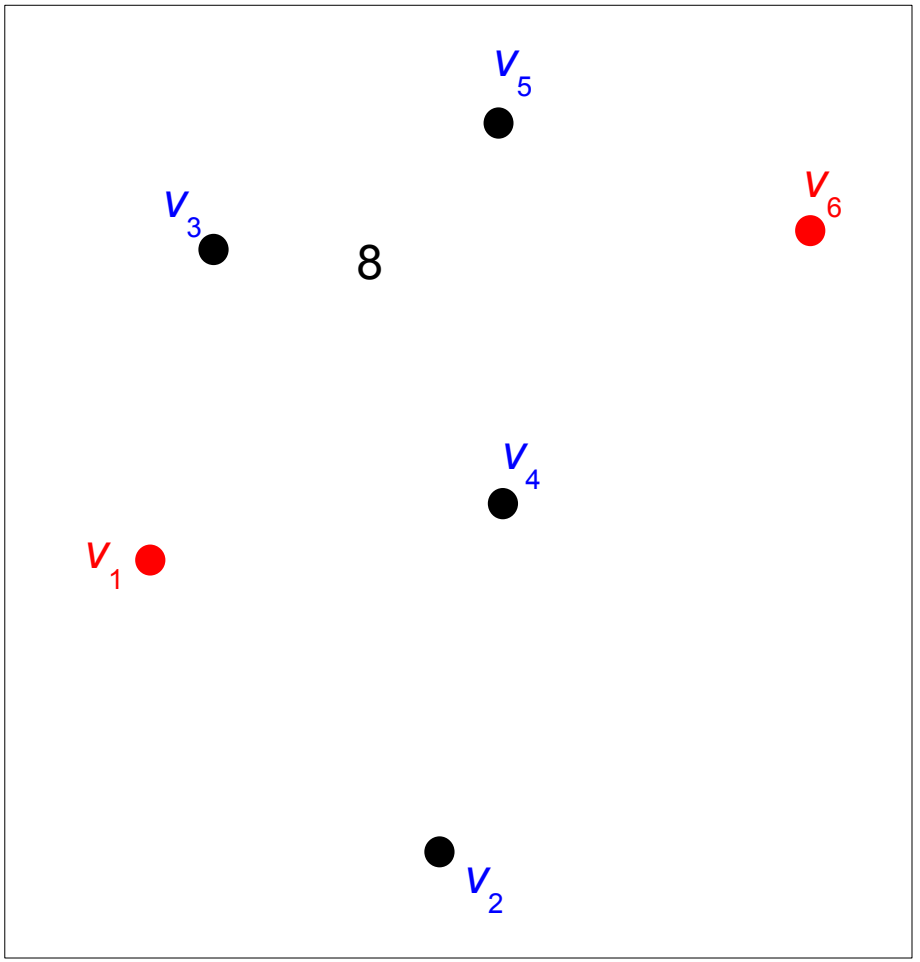
Przepływy:

$F(v_1, v_2, v_6) = \mathbf{5}$

$F(v_1, v_3, v_5, v_6) = \mathbf{2}$

$F(v_1, v_4, v_2, v_6) = \mathbf{2}$

Algorytm FORDA-FULKERSONA



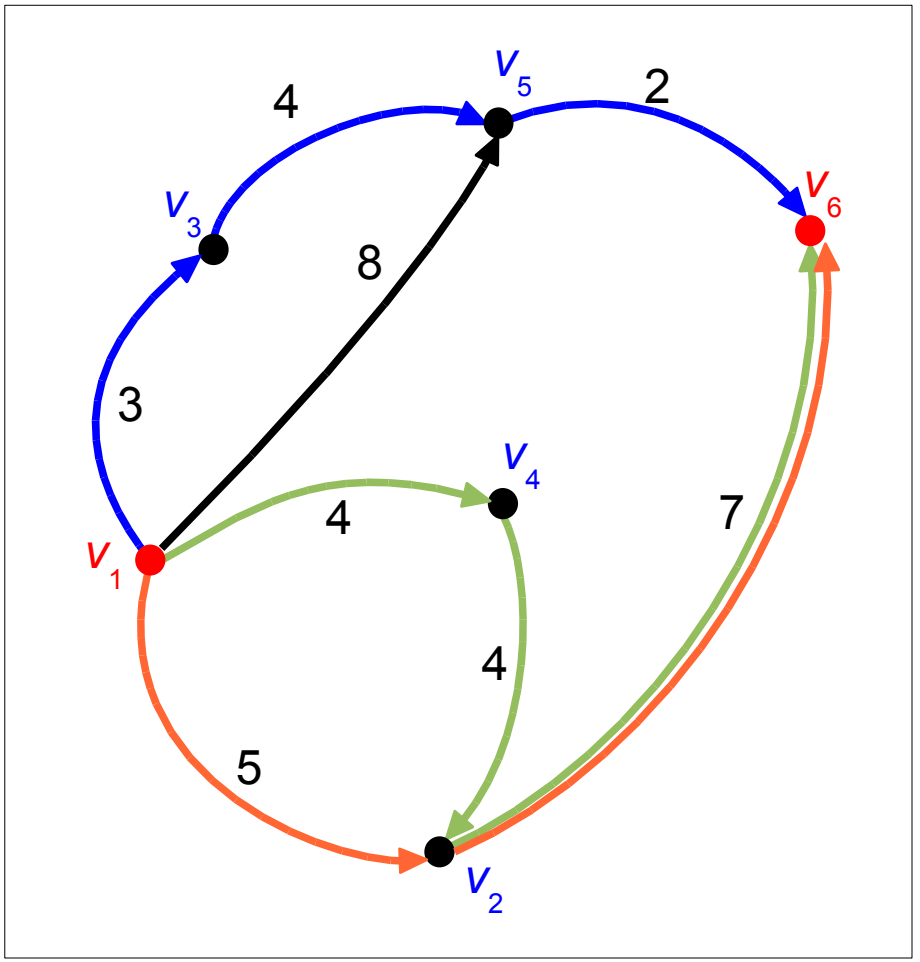
Przepływy:

$F(v_1, v_2, v_6) = \mathbf{5}$

$F(v_1, v_3, v_5, v_6) = \mathbf{2}$

$F(v_1, v_4, v_2, v_6) = \mathbf{2}$

Algorytm FORDA-FULKERSONA



Przepływy:

$F(v_1,v_2,v_6) = \mathbf{5}$

$F(v_1,v_3,v_5,v_6) = \mathbf{2}$

$F(v_1,v_4,v_2,v_6) = \mathbf{2}$

Maks. przepływ = 9