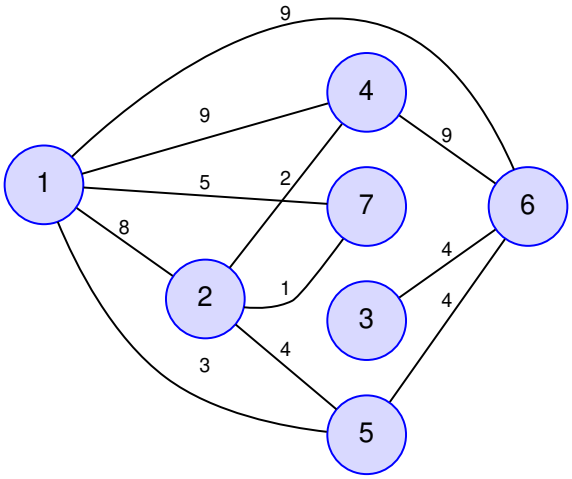
Omówienie – zestaw 3.[1]

Ad. 1. Spójny nieskierowany graf losowy z wagami

Pierwsze zadanie polega na wygenerowaniu losowego spójnego grafu prostego. W tym celu wystarczy wykorzystać algorytmy zaimplementowane w poprzednich projektach. Nowością są **wagi**: każdej krawędzi wygenerowanego grafu należy przypisać losową wagę z przedziału $[1, 10]$. W tym zestawie będziemy zajmować się **najkrótszymi ścieżkami** pomiędzy wierzchołkami, przy czym długość ścieżki rozumiemy jako **sumę wag krawędzi**, z których ścieżka się składa. Wobec tego sama liczba krawędzi, wchodzących w skład danej ścieżki, nie jest tutaj istotna. Dalszy opis projektu będzie zakładał, że pracujemy na grafie G z rys. 1.

Uwaga: treść zadanie wymaga, aby program generował graf losowy i na nim pracował. Mimo to zalecam, aby program mógł również pracować na zadanym wejściowym grafie, żeby mogli Państwo wykonać test np. na przykładzie z udostępnionych Państwu materiałów/z internetu, itp. Nie będzie to oceniane, ale może być dla Państwa bardzo pomocne.



Rysunek 1: Przykładowy spójny losowy graf prosty G z wagami z przedziału $[1, 10]$. Wagi będą oznaczane jako w, gdzie $w(u, v)$ jest wagą krawędzi (u, v)

Ad. 2. Algorytm Dijkstry (*Algorytm wraz z algorytmami pomocniczymi został udostępniony na UPeL-u w osobnym pliku: "Algorytmy do projektu nr 3"*)

W 2. zadaniu naszym celem jest **znalezienie najkrótszych ścieżek od zadanego wierzchołka startowego** s (tutaj wybrano: $s = 1$) do wszystkich pozostałych wierzchołków przy pomocy **algorytmu Dijkstry**.

Algorytm rozpoczyna się przygotowaniem dwóch **tablic atrybutów**: d_s oraz p_s. Obie tablice powinny mieć tyle elementów, ile graf ma wierzchołków. Znaczenie elementów – dla wierzchołka v:

- $d_s(v)$ $\equiv$ **oszacowanie wagi** najkrótszej ścieżki od wierzchołka s do v, tj. **górne ograniczenie** wagi tej ścieżki. Algorytm będzie stale poprawiał wartości tych elementów – tak, że po

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

zakończeniu działania algorytmu $d_s(v)$ będzie **faktyczną wagą (długością) najkrótszej ścieżki od** s **do** v.

- $p_s(v)$ $\equiv$ **poprzednik** wierzchołka v w najkrótszej ścieżce z s do v. Po zakończeniu działania algorytmu będziemy znali poprzednika każdego z wierzchołków. Ta wiedza pozwoli na odtworzenie najkrótszej ścieżki z wierzchołka s do każdego pozostałego wierzchołka.

Algorytm pomocniczy INIT(G, s): **Początkowo** dla każdego z wierzchołków w tablicy d_s przypisujemy **nieskończoność** (zawsze prawidłowe górne ograniczenie wagi najkrótszej ścieżki), natomiast w tablicy p_s: `Nil/Null` jako informację o (chwilowym) braku poprzednika. Ponadto dla **wierzchołka startowego** s uzupełniamy: $d_s(s) = 0$, ponieważ wiadomo, że taka będzie długość najkrótszej ścieżki od s do s.

Tworzymy zbiór "gotowych" wierzchołków S, który początkowo jest pusty. Stopniowo będziemy dodawać do niego te wierzchołki, których atrybuty zostały już uznane przez algorytm jako ostateczne.

Algorytm Dijkstry będzie wykonywał tyle iteracji, ile będzie potrzebnych, aby w zbiorze S znalazły się **wszystkie wierzchołki**. **Każda z iteracji** będzie polegała na:

- dodaniu do zbioru S takiego wierzchołka spośród nienależących do S, który ma najmniejszy atrybut $d_s(u)$ – "**akceptacja wierzchołka**" $\rightarrow$ uznajemy, że $d_s(u)$ oraz $p_s(u)$ są ostateczną długością najkrótszej ścieżki oraz poprzednikiem w najkrótszej ścieżce od s do u;
- **relaksacji każdej krawędzi** z u do tych wierzchołków, które jeszcze nie należą do zbioru S. Korzystamy tu z algorytmu pomocniczego RELAX(u, v, w), gdzie v oznacza wierzchołek incydentny z krawędzią (u, v), dla której wykonujemy relaksację. Relaksacja polega na sprawdzeniu, czy aktualne oszacowanie długości $d_s(v)$ jest większe (gorsze), niż oszacowanie prowadzące do wierzchołka v przez wierzchołek u (patrz: algorytm relaksacji w dokumencie "Algorytmy do projektu nr 3").

Kolejne kroki algorytmu dla przykładowego grafu z rys. 1 zostały przedstawione i omówione w tabeli 1.

$d_1(1) = 0$	———
$d_1(2) = \infty$	$p_1(2) = \texttt{Nil}$
$d_1(3) = \infty$	$p_1(3) = \texttt{Nil}$
$d_1(4) = \infty$	$p_1(4) = \texttt{Nil}$
$d_1(5) = \infty$	$p_1(5) = \texttt{Nil}$
$d_1(6) = \infty$	$p_1(6) = \texttt{Nil}$
$d_1(7) = \infty$	$p_1(7) = \texttt{Nil}$

(a) Stan po wykonaniu algorytmu pomocniczego INIT(G, s) dla $s = 1$.

$d_1(1) = 0$	———
$d_1(2) = \min\big(\infty, d_1(1) + 8\big) = 8$	$p_1(2) = 1$
$d_1(3) = \infty$	$p_1(3) = \texttt{Nil}$
$d_1(4) = \min\big(\infty, d_1(1) + 9\big) = 9$	$p_1(4) = 1$
$d_1(5) = \min\big(\infty, d_1(1) + 3\big) = 3$	$p_1(5) = 1$
$d_1(6) = \min\big(\infty, d_1(1) + 9\big) = 9$	$p_1(6) = 1$
$d_1(7) = \min\big(\infty, d_1(1) + 5\big) = 5$	$p_1(7) = 1$

(b) **1. iteracja algorytmu Dijkstry**: do zbioru S dołączono wierzchołek nr 1. Następuje relaksacja krawędzi $(1, 2)$, $(1, 4)$, $(1, 5)$, $(1, 6)$ oraz $(1, 7)$. Podczas relaksacji każdej z krawędzi $(1, v)$ nowe oszacowanie $d_s(v)$ zostaje wyznaczone jako $\min\big(d_s(v), d_s(1) + w(1, v)\big)$.

$d_1(1) = 0$	
$d_1(2) = \min\left(8, d_1(5) + 4\right) = 7$	$p_1(2) = 5$
$d_1(3) = \infty$	$p_1(3) = \texttt{Nil}$
$d_1(4) = 9$	$p_1(4) = 1$
$d_1(5) = 3$	$p_1(5) = 1$
$d_1(6) = \min\left(9, d_1(5) + 4\right) = 7$	$p_1(6) = 5$
$d_1(7) = 5$	$p_1(7) = 1$

(c) **2. iteracja algorytmu Dijkstry**: do zbioru S dołączono wierzchołek nr 5. Następuje relaksacja krawędzi $(5, 2)$ oraz $(5, 6)$.

$d_1(1) = 0$	
$d_1(2) = \min\left(7, d_1(7) + 1\right) = 6$	$p_1(2) = 7$
$d_1(3) = \infty$	$p_1(3) = \texttt{Nil}$
$d_1(4) = 9$	$p_1(4) = 1$
$d_1(5) = 3$	$p_1(5) = 1$
$d_1(6) = 7$	$p_1(6) = 5$
$d_1(7) = 5$	$p_1(7) = 1$

(d) **3. iteracja algorytmu Dijkstry**: do zbioru S dołączono wierzchołek nr 7. Następuje relaksacja krawędzi $(7, 2)$.

$d_1(1) = 0$	
$d_1(2) = 6$	$p_1(2) = 7$
$d_1(3) = \infty$	$p_1(3) = \texttt{Nil}$
$d_1(4) = \min\left(9, d_1(2) + 2\right) = 8$	$p_1(4) = 2$
$d_1(5) = 3$	$p_1(5) = 1$
$d_1(6) = 7$	$p_1(6) = 5$
$d_1(7) = 5$	$p_1(7) = 1$

(e) **4. iteracja algorytmu Dijkstry**: do zbioru S dołączono wierzchołek nr 2. Następuje relaksacja krawędzi $(2, 4)$.

$d_1(1) = 0$	
$d_1(2) = 6$	$p_1(2) = 7$
$d_1(3) = \min\left(\infty, d_1(6) + 4\right) = 11$	$p_1(3) = 6$
$d_1(4) = \min\left(8, d_1(6) + 9\right) = 8$	$p_1(4) = 2$
$d_1(5) = 3$	$p_1(5) = 1$
$d_1(6) = 7$	$p_1(6) = 5$
$d_1(7) = 5$	$p_1(7) = 1$

(f) **5. iteracja algorytmu Dijkstry**: do zbioru S dołączono wierzchołek nr 6. Następuje relaksacja krawędzi $(6, 3)$ oraz $(6, 4)$.

$d_1(1) = 0$	
$d_1(2) = 6$	$p_1(2) = 7$
$d_1(3) = 11$	$p_1(3) = 6$
$d_1(4) = 8$	$p_1(4) = 2$
$d_1(5) = 3$	$p_1(5) = 1$
$d_1(6) = 7$	$p_1(6) = 5$
$d_1(7) = 5$	$p_1(7) = 1$

(g) **6. iteracja algorytmu Dijkstry**: do zbioru S dołączono wierzchołek nr 4. Brak relaksacji (wierzchołek $u = 4$ nie ma sąsiadów, którzy nie należą do S).

$d_1(1) = 0$	
$d_1(2) = 6$	$p_1(2) = 7$
$d_1(3) = 11$	$p_1(3) = 6$
$d_1(4) = 8$	$p_1(4) = 2$
$d_1(5) = 3$	$p_1(5) = 1$
$d_1(6) = 7$	$p_1(6) = 5$
$d_1(7) = 5$	$p_1(7) = 1$

(h) **7. iteracja algorytmu Dijkstry**: do zbioru S dołączono wierzchołek nr 3. Brak relaksacji (wierzchołek $u = 4$ nie ma sąsiadów, którzy nie należą do S). Koniec algorytmu.

Tabela 1: Kolejne kroki algorytmu Dijkstry dla grafu wejściowego z rys. 1. Zielonym kolorem zaznaczono atrybuty tych wierzchołków, które w danej iteracji algorytmu należą do zbioru S (na początku każdej iteracji jeden wierzchołek staje się 'gotowym" u). Niebieskim kolorem zapisano atrybuty wierzchołków v incydentnych z krawędziami (u, v), które w danej iteracji zostały poddane relaksacji.

Podsumowując: w wyniku działania algorytmu Dijkstry dla wierzchołka startowego $s = 1$ uzyskano najkrótsze ścieżki, które zaprezentowano na listingu 1.

```
START: s = 1
d(1) = 0 ==> [1]
d(2) = 6 ==> [1, 7, 2]
d(3) = 11 ==> [1, 5, 6, 3]
d(4) = 8 ==> [1, 7, 2, 4]
d(5) = 3 ==> [1, 5]
d(6) = 7 ==> [1, 5, 6]
d(7) = 5 ==> [1, 7]
```

Listing 1: Najkrótsze ścieżki dla grafu z rys. 1. Wierzchołek startowy: `s = 1`; `d(v)` – długość najkrótszej ścieżki od wierzchołka `s` do wierzchołka `v`. Obok długości wypisano ścieżkę w formie numerów poszczególnych wierzchołków, które ją tworzą.

Ad. 3. Macierz odległości

W kolejnym zadaniu należy wyznaczyć **macierz odległości** pomiędzy wszystkimi parami wierzchołków w naszym grafie. W tym celu wystarczy wywołać algorytm Dijkstry tyle razy, ile w grafie jest wierzchołków: za każdym razem dla innego wierzchołka startowego s. Wynikowa tablica d_s będzie s-tym wierszem macierzy odległości.

W wersji zoptymalizowanej każde kolejne wywołanie algorytmu Dijkstry może korzystać z gotowych długości, które znaleziono wcześniej: np. zakładając, że wywołano już algorytm Dijkstry dla $s = 1$, w wywołaniu dla $s = 2$ można ominąć wyszukiwanie najkrótszej ścieżki od wierzchołka 2 do 1. Korzystając z faktu, że graf jest nieskierowany, wiemy, że długość tej ścieżki jest taka sama, jak w przypadku od wierzchołka 1 do 2. Wniosek: wynikowa macierz musi być **symetryczna** względem diagonali (np.: listing 2.).

```
0    6   11    8    3    7    5
6    0   12    2    4    8    1
11  12    0   13    8    4   13
8    2   13    0    6    9    3
3    4    8    6    0    4    5
7    8    4    9    4    0    9
5    1   13    3    5    9    0
```

Listing 2: Macierz odległości dla grafu z rys. 1; element w wierszu `w` i kolumnie `k` oznacza długość najkrótszej ścieżki od wierzchołka `w` do `k`.

Ad. 4. Centrum/centrum minimax

Treść zadania dokładnie tłumaczy, jak definiujemy **centrum** oraz **centrum minimax**:

- **centrum grafu** "to wierzchołek, którego suma odległości do pozostałych wierzchołków jest minimalna";
- **centrum minimax** "to wierzchołek, którego odległość do najdalszego wierzchołka jest minimalna".

Oba centra wyznaczamy na podstawie macierzy odległości z 3. zadania. Dla omawianego grafu z rys. 1 centrum grafu oraz centrum minimax to **wierzchołek nr 5** (patrz: tabela 2.).

Uwaga: dla przykładowego grafu z rys. 1 okazało się, że centrum oraz centrum minimax to ten sam wierzchołek, ale oczywiście nie dla każdego grafu tak będzie – mogą to być dwa różne wierzchołki.

Numer wierzchołka v	Suma odległości do pozostałych wierzchołków	Odległość do najdalszego wierzchołka
1	40	11
2	33	12
3	61	13
4	41	13
5	**30**	**8**
6	41	9
7	36	13

Tabela 2: Suma odległości do pozostałych wierzchołków oraz odległość do najdalszego wierzchołka dla grafu z rys. 1., wyznaczone na podstawie macierzy odległości z listingu 2.

Ad. 5. Minimalne drzewo rozpinające

Drzewem nazywamy **spójny graf prosty bez cykli**. Graf, w którym nie ma cykli, ale nie jest spójny, nazywamy **lasem**. Wynika stąd, że **las** składa się z **drzew**.

Z definicji drzewa wynika, że **każda para wierzchołków** jest połączona **dokładnie jedną drogą**. Jeżeli drzewo składa się z n wierzchołków, to ma $n-1$ krawędzi. Każda krawędź drzewa jest **mostem**.

Drzewo rozpinające grafu G składa się z każdego wierzchołka tego grafu oraz jego niektórych krawędzi. Krawędzie są wybrane w taki sposób, by drzewo było **acykliczne i spójne** (co wynika z samej definicji drzewa). Dla danego grafu G najczęściej istnieje więcej niż jedno drzewo rozpinające.

Minimalne drzewo rozpinające grafu G to drzewo rozpinające o najmniejszej sumie wag krawędzi należących do drzewa.

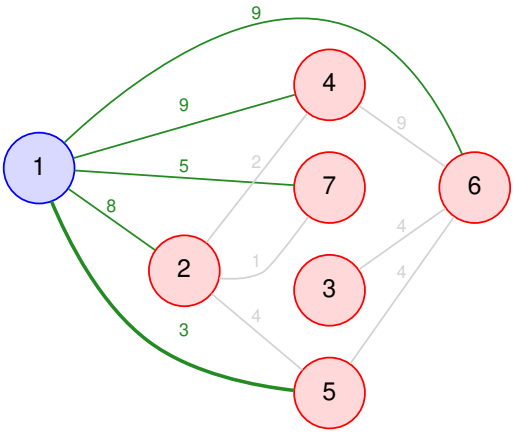
Zgodnie z treścią zadania, należy zaimplementować jeden z dwóch algorytmów wyznaczania minimalnego drzewa rozpinającego. Poniżej zostaną omówione oba niezależne algorytmy. Algorytmy zostały przedstawione również w dokumencie "Algorytmy do projektu nr 3".

Algorytm Prima

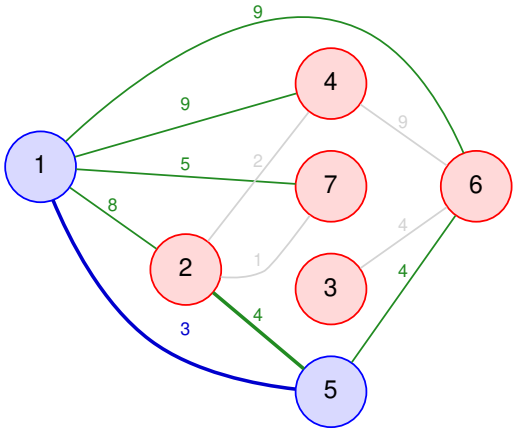
Wierzchołki grafu dzielimy na **dwie części**: dowolny wierzchołek startowy dodajemy do pustego drzewa T, a pozostałe wierzchołki tworzą zbiór W. W każdej iteracji algorytmu sprawdzamy tylko te krawędzie, które łączą T z W: spośród nich wybieramy **krawędź lekką** i dodajemy ją do T, usuwając odpowiedni wierzchołek z W. **Krawędź lekka** to ta krawędź, która ma najmniejszą wagę spośród sprawdzanych w danej iteracji krawędzi.

Algorytm kończy się, gdy T zawiera już wszystkie wierzchołki grafu.

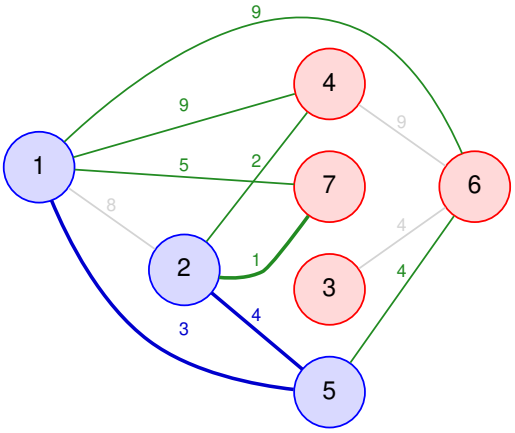
Poszczególne kroki algorytmu dla przykładowego grafu z rys. 1 zaprezentowano na wykresach z 2(a)–2(g) przy założeniu, że wystartowano od wierzchołka nr 1.



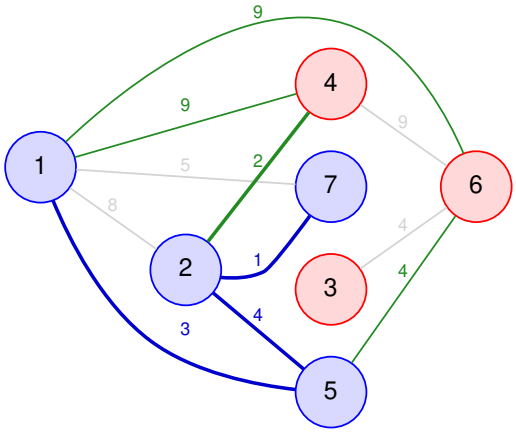
(a) **1. krok algorytmu Prima**: do T początkowo należy tylko wierzchołek startowy (nr 1); krawędzią lekką jest $(1, 5)$, która zostaje dodana do T.



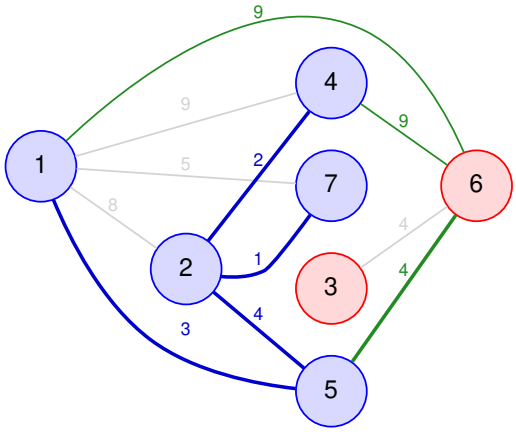
(b) **2. krok algorytmu Prima**: do T należą wierzchołki: 1 i 2; krawędzią lekką jest $(2, 5)$.



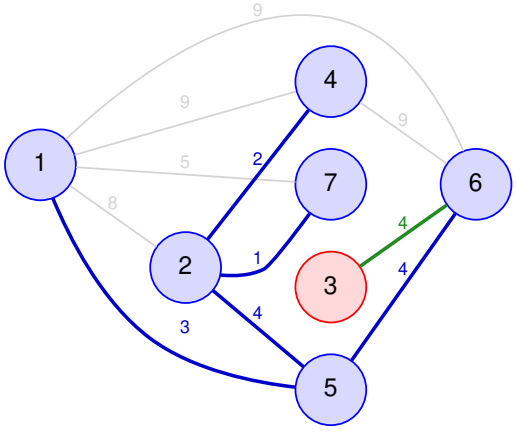
(c) **3. krok algorytmu Prima**: krawędzią lekką jest $(2, 7)$.



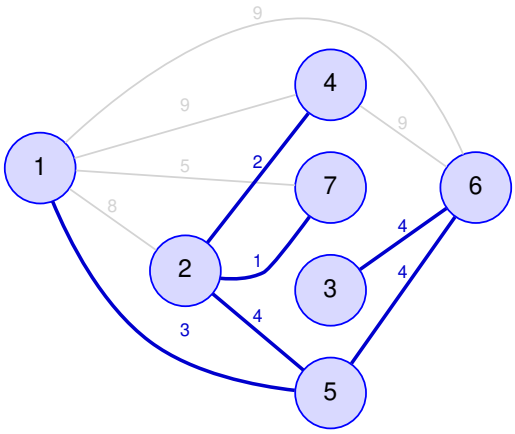
(d) **4. krok algorytmu Prima**: krawędzią lekką jest $(2, 4)$.



(e) **5. krok algorytmu Prima**: krawędzią lekką jest $(5, 6)$.



(f) **6. krok algorytmu Prima**: krawędzią lekką jest $(3, 6)$. Po dodaniu tej krawędzi do T, drzewo T zawiera już wszystkie wierzchołki grafu: koniec działania algorytmu.



(g) Minimalne drzewo rozpinające grafu z rys. 1, uzyskane przy pomocy **algorytmu Prima**. Do drzewa należą tylko pogrubione, niebieskie krawędzie.

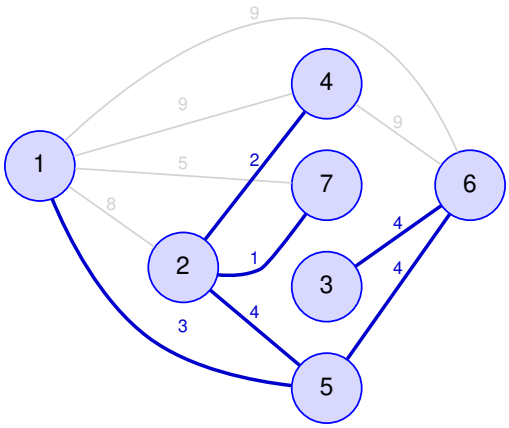
Rysunek 2: Działanie algorytmu Prima dla grafu z rys. 1. Niebieskie wierzchołki i krawędzie należą do drzewa T w danej iteracji algorytmu. Czerwone wierzchołki należą do zbioru W. Krawędzie łączące T z W w danym kroku są zaznaczone na zielono.

Algorytm Kruskala

Do drzewa T będą należały wszystkie wierzchołki: początkowo nie są połączone. **Sortujemy krawędzie grafu G według niemalejących wag**. Następnie w tej kolejności sprawdzamy, czy dodanie danej krawędzi do T spowoduje powstanie cyklu: jeśli nie, to dodajemy ją do T. Algorytm można przerwać, gdy T składa się już z $n-1$ krawędzi (n jest liczbą wierzchołków grafu G). Poszczególne kroki algorytmu dla grafu z rys. 1. przedstawiono w tabeli 3.

Krawędź (u,v)	Waga krawędzi $w(u,v)$	Przynależność krawędzi (u,v) do drzewa T
$(2,7)$	1	Tak
$(2,4)$	2	Tak
$(1,5)$	3	Tak
$(2,5)$	4	Tak
$(3,6)$	4	Tak
$(5,6)$	4	Tak
$(1,7)$	5	Nie
$(1,2)$	8	Nie
$(1,4)$	9	Nie
$(1,6)$	9	Nie
$(4,6)$	9	Nie

Tabela 3: Działanie algorytmu Kruskala dla grafu z rys. 1. Krawędzie wypisano w kolejności od najmniejszej wagi. *Uwaga: w tym przypadku okazało się, że pierwszych $n-1$ krawędzi należy do drzewa T, ale nie zawsze tak będzie: w ogólności trzeba omijać krawędzie, które spowodują powstanie cyklu.*



Rysunek 3: Minimalne drzewo rozpinające grafu z rys. 1, uzyskane przy pomocy **algorytmu Kruskala**; do drzewa należą tylko pogrubione, niebieskie krawędzie.

Oczywiście oba algorytmy prowadzą do znalezienia tego samego minimalnego drzewa rozpinającego (wyjątkiem mógłby być taki graf, w którym istnieje więcej niż jedno drzewo o tej samej, minimalnej sumie wag).