Algorytmy i struktury danych

Grafy

Krzysztof M. Ocetkiewicz
Krzysztof.Ocetkiewicz@eti.pg.edu.pl

Katedra Algorytmów i Modelowania Systemów, WETI, PG

Grafy

- graf to zbiór wierzchołków i zbiór krawędzi
- każda krawędź łączy dokładnie dwa wierzchołki (poza pętlami, które łączą wierzchołek z samym sobą)
- graf prosty – nie posiada pętli i krawędzi wielokrotnych
- graf skierowany – każda krawędź jest skierowana (ma określony kierunek) – krawędź skierowana z A do B łączy A z B ale nie B z A (np. ulica jednokierunkowa)

Grafy

- wierzchołki sąsiednie – wierzchołki połączone krawędzią
- wierzchołek sąsiaduje z krawędzią, jeżeli jest jej końcem
- ścieżka – ciąg sąsiadujących wierzchołków (nie powtarzających się) i łączących je krawędzi
- graf jest spójny, jeżeli dla każdej pary wierzchołków istnieje ścieżka je łącząca
- spójna składowa – maksymalny spójny podgraf grafu

Grafy

- n – liczba wierzchołków w grafie
- m – liczba krawędzi w grafie
- zarówno wierzchołki jak i krawędzie mogą mieć etykiety (dodatkowe informacje)

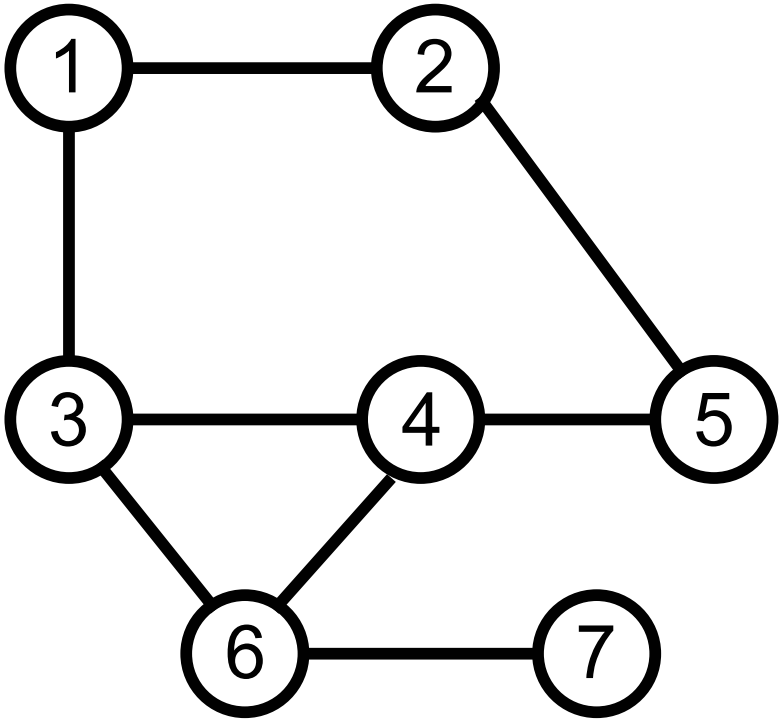
Reprezentacje grafu w pamięci

- macierz sąsiedztwa
- listy sąsiedztwa
- macierz koincydencji
- lista krawędzi
- ...

Macierz sąsiedztwa

- macierz kwadratowa $n \cdot n$
- $M_{u,v}$ opisuje krawędź pomiędzy wierzchołkami u i v
- np. 1 – jest krawędź, 0 – nie ma krawędzi
- np. -1 – nie ma krawędzi, wszystko inne – etykieta krawędzi

Macierz sąsiedztwa

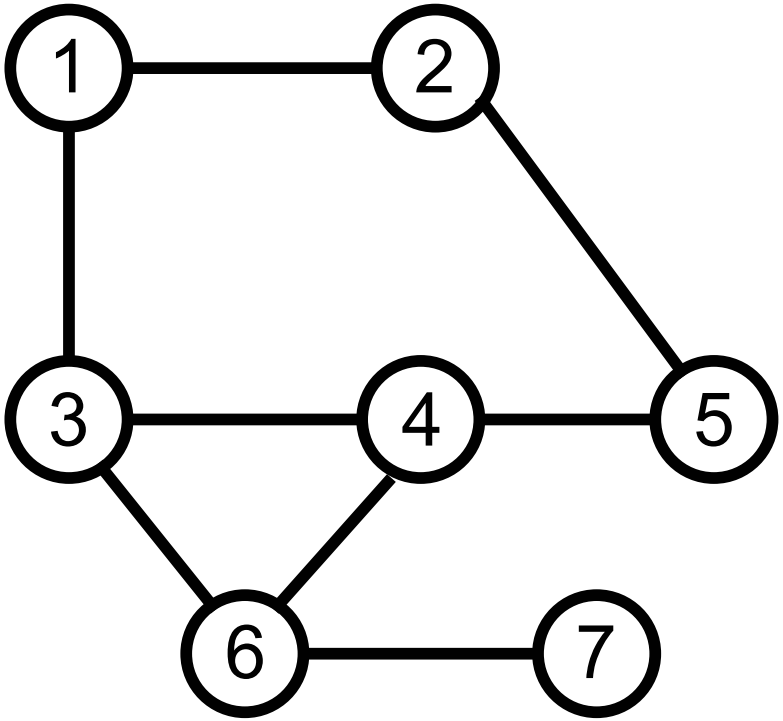


	1	2	3	4	5	6	7
1	0	1	1	0	0	0	0
2	1	0	0	0	1	0	0
3	1	0	0	1	0	1	0
4	0	0	1	0	1	1	0
5	0	1	0	1	0	0	0
6	0	0	1	1	0	0	1
7	0	0	0	0	0	1	0

Macierz sąsiedztwa

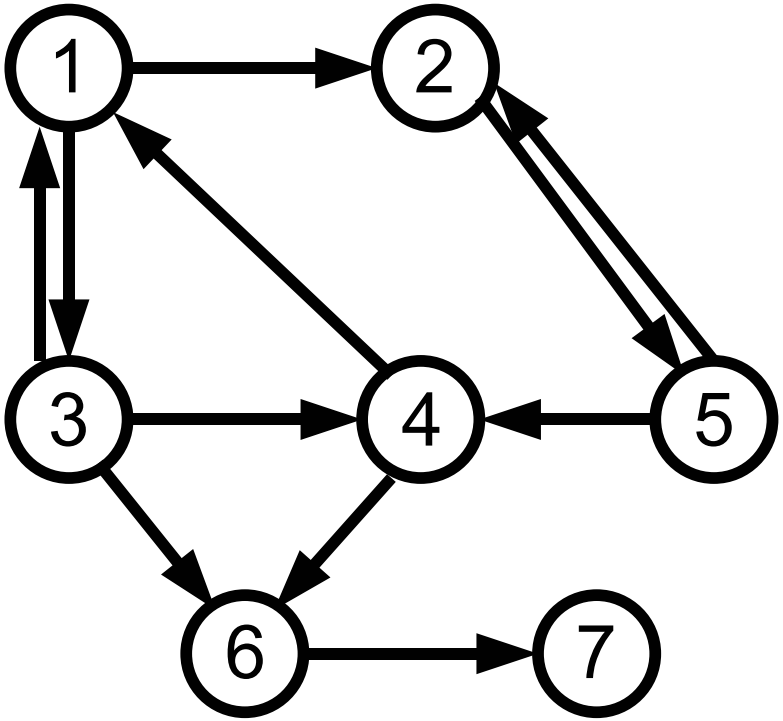
- do przedstawienia grafu nieskierowanego wystarczy macierz trójkątna (macierz sąsiedztwa jest symetryczna: $M_{u,v} = M_{v,u}$)

Macierz sąsiedztwa



	1	2	3	4	5	6	7
1	0						
2	1	0					
3	1	0	0				
4	0	0	1	0			
5	0	1	0	1	0		
6	0	0	1	1	0	0	
7	0	0	0	0	0	1	0

Macierz sąsiedztwa



	1	2	3	4	5	6	7
1	0	1	1	0	0	0	0
2	0	0	0	0	1	0	0
3	1	0	0	1	0	1	0
4	1	0	0	0	0	1	0
5	0	1	0	1	0	0	0
6	0	0	0	0	0	0	1
7	0	0	0	0	0	0	0

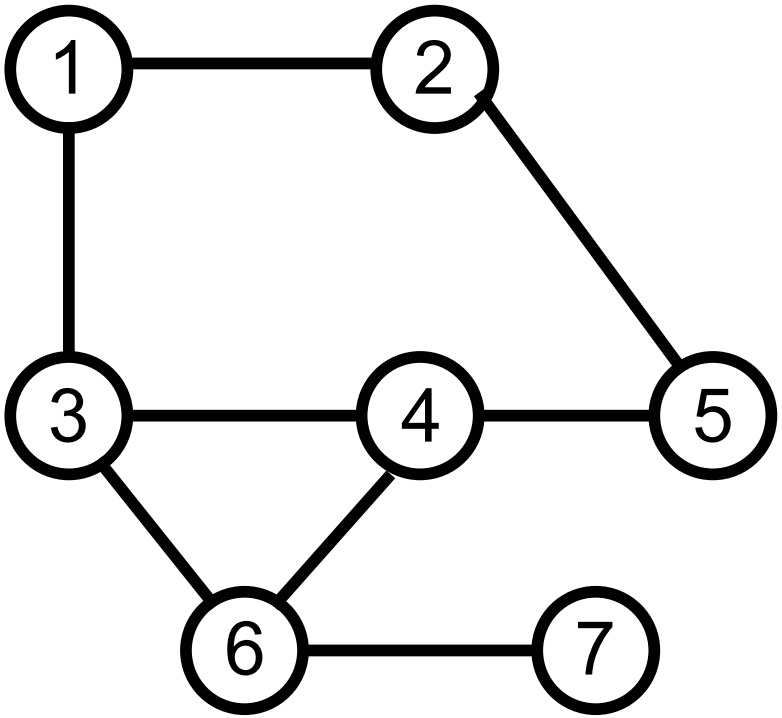
Macierz sąsiedztwa

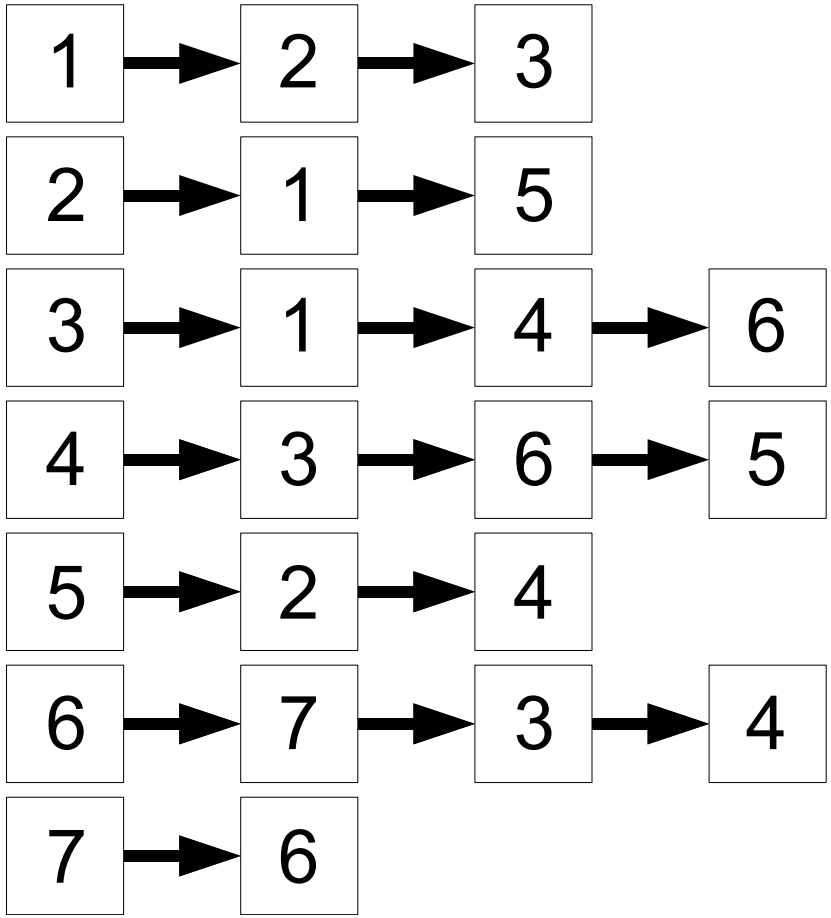
- szybkie sprawdzanie istnienia, wstawianie, usuwanie krawędzi pomiędzy u i v
- szybkie pobranie etykiety krawędzi pomiędzy u i v
- wolne przeglądanie sąsiedztwa (istotne przy grafach rzadkich)
- duże wymagania pamięciowe

Listy sąsiedztwa

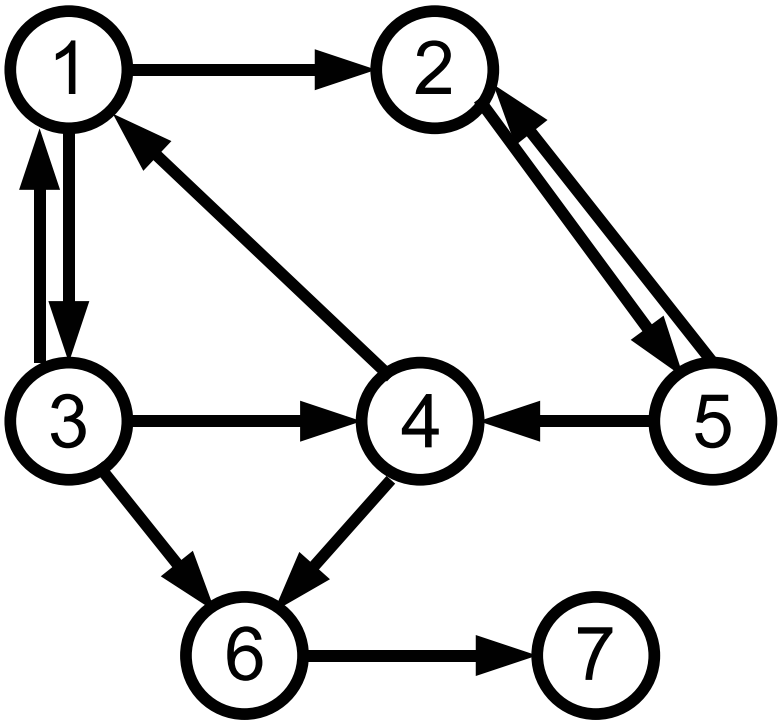
- n list zawierających numery lub wskaźniki na sąsiadów danego wierzchołka
- w grafie nieskierowanym, jeżeli v sąsiaduje z u to v jest na liście u i u jest na liście v

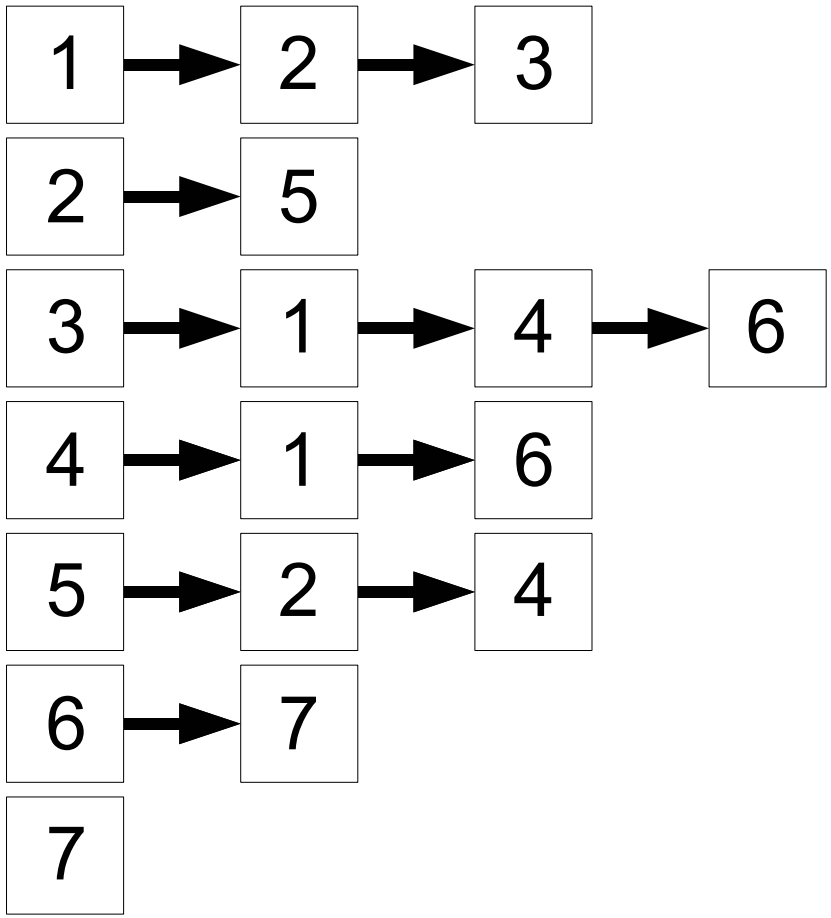
Listy sąsiedztwa





Listy sąsiedztwa





Listy sąsiedztwa

- wolne sprawdzanie istnienia i usuwanie krawędzi pomiędzy u i v
- wolne pobranie etykiety krawędzi pomiędzy u i v
- szybkie wstawianie krawędzi pomiędzy u i v
- szybkie przeglądanie sąsiedztwa
- mniejsze wymagania pamięciowe

Macierz koincydencji

- macierz o wymiarach n na m
- $M_{v,e}$ opisuje połączenie pomiędzy wierzchołkiem v a krawędzią e (np. 1 – jest, 0 – nie ma)
- w przypadku grafów skierowanych np. -1 – krawędź wychodzi z v, 0 – brak połączenia, 1 – krawędź wchodzi do v

Przeglądanie w głąb

- algorytm odwiedzania wszystkich wierzchołków z jednej spójnej składowej
- zaczynamy od dowolnego wierzchołka – odwiedzimy całą spójną składową, do której on należy
- każdy wierzchołek jest albo odwiedzony albo nieodwiedzony
- dla każdego przetwarzanego wierzchołka odkładamy na stos jego nieodwiedzonych sąsiadów i oznaczamy ich jako odwiedzonych
- przechodzimy do pierwszego nieodwiedzonego wierzchołka, jeżeli się nie da, wycofujemy się

Przeglądanie w głąb

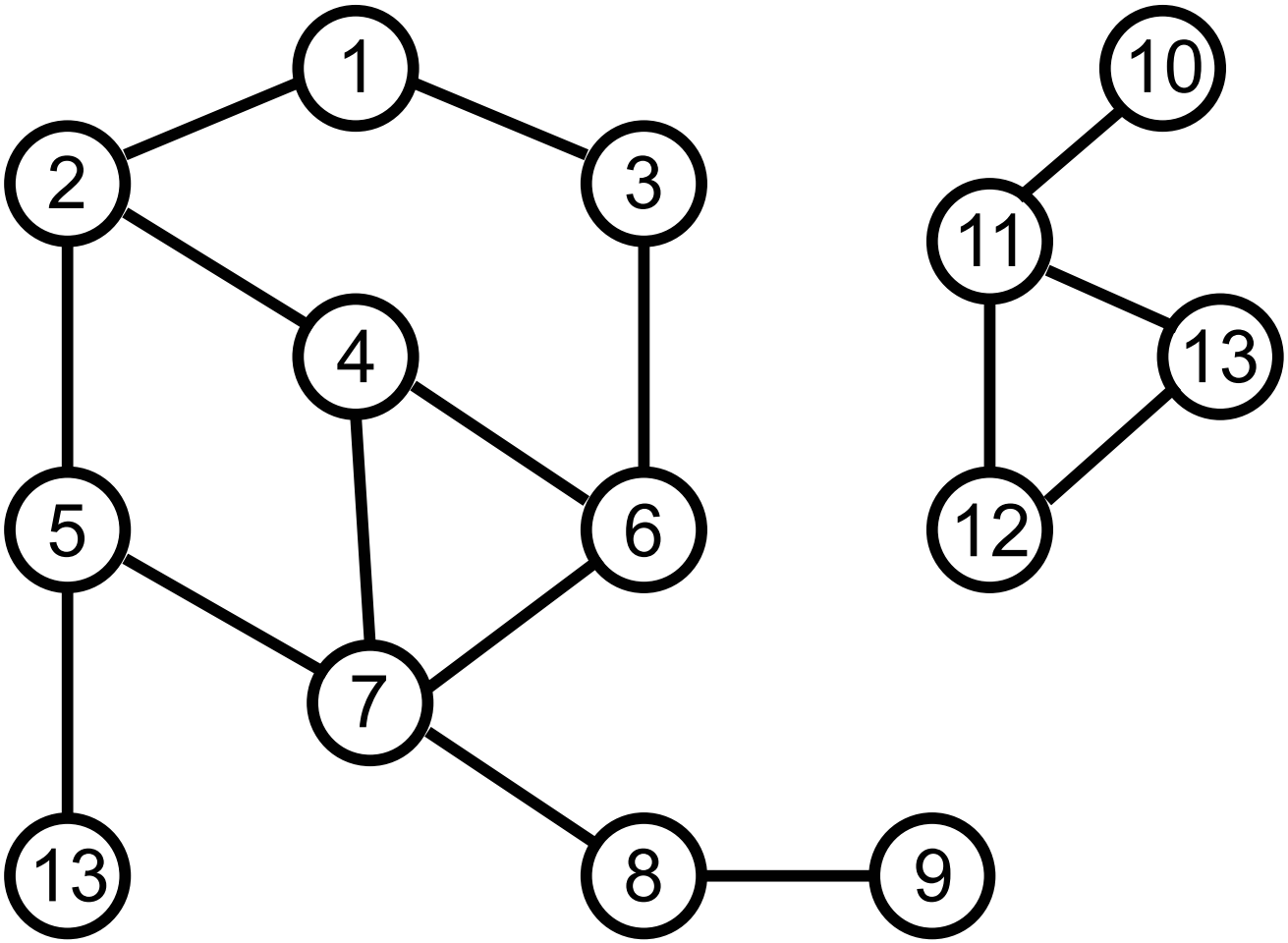
- funkcja rekurencyjna:

  $Odwiedz(v, odw)$:
  $odw[v] = True$
  for $u =$ każdy sąsiad v **do**
  if $odw[u]$ **then continue**
  $Odwiedz(u, odw)$
  end for

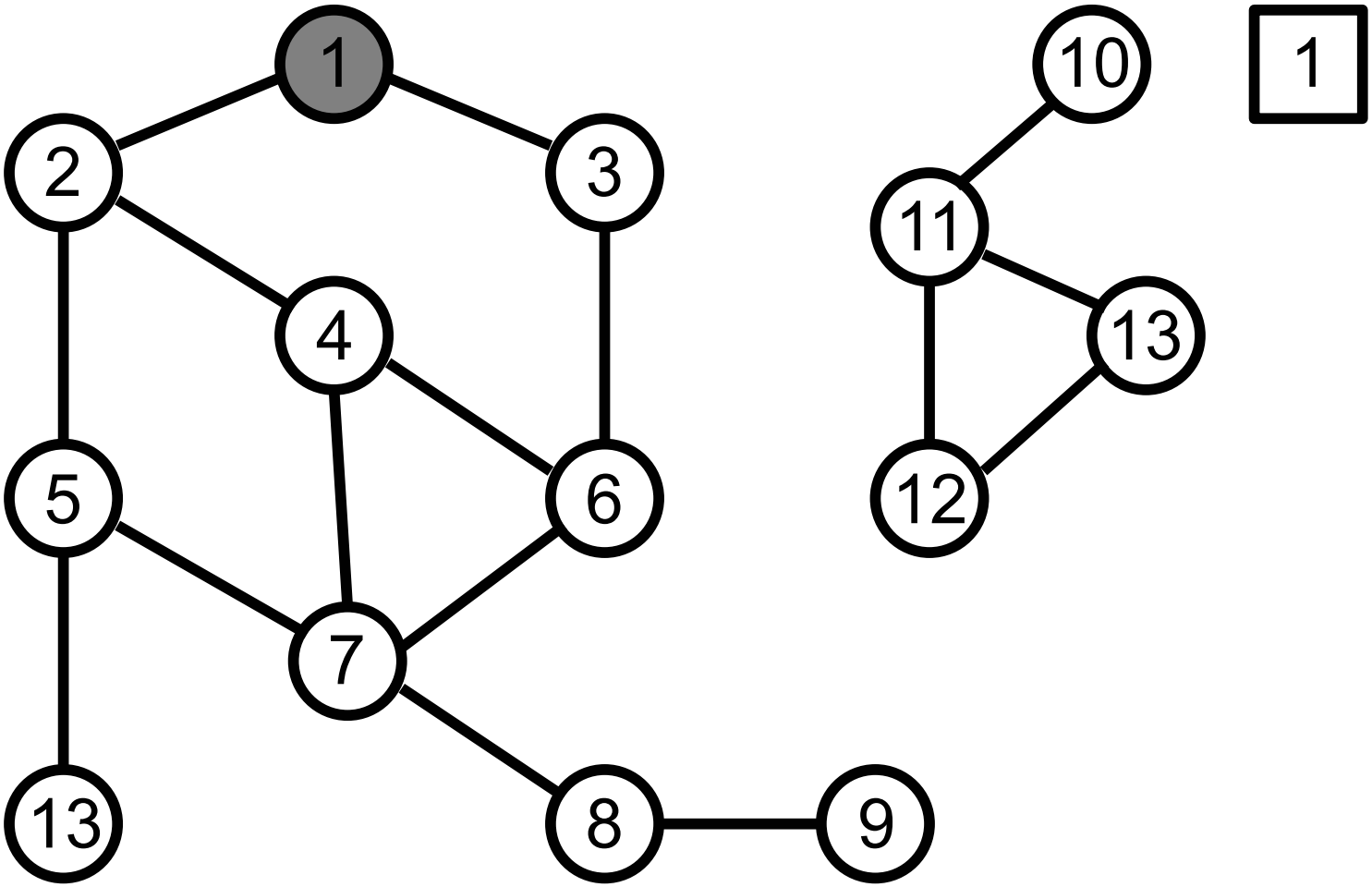
- przeglądanie w głąb:

  for $i = 1, \ldots, n$ **do** $odw[i] = False$
  $Odwiedz(v_s, odw)$

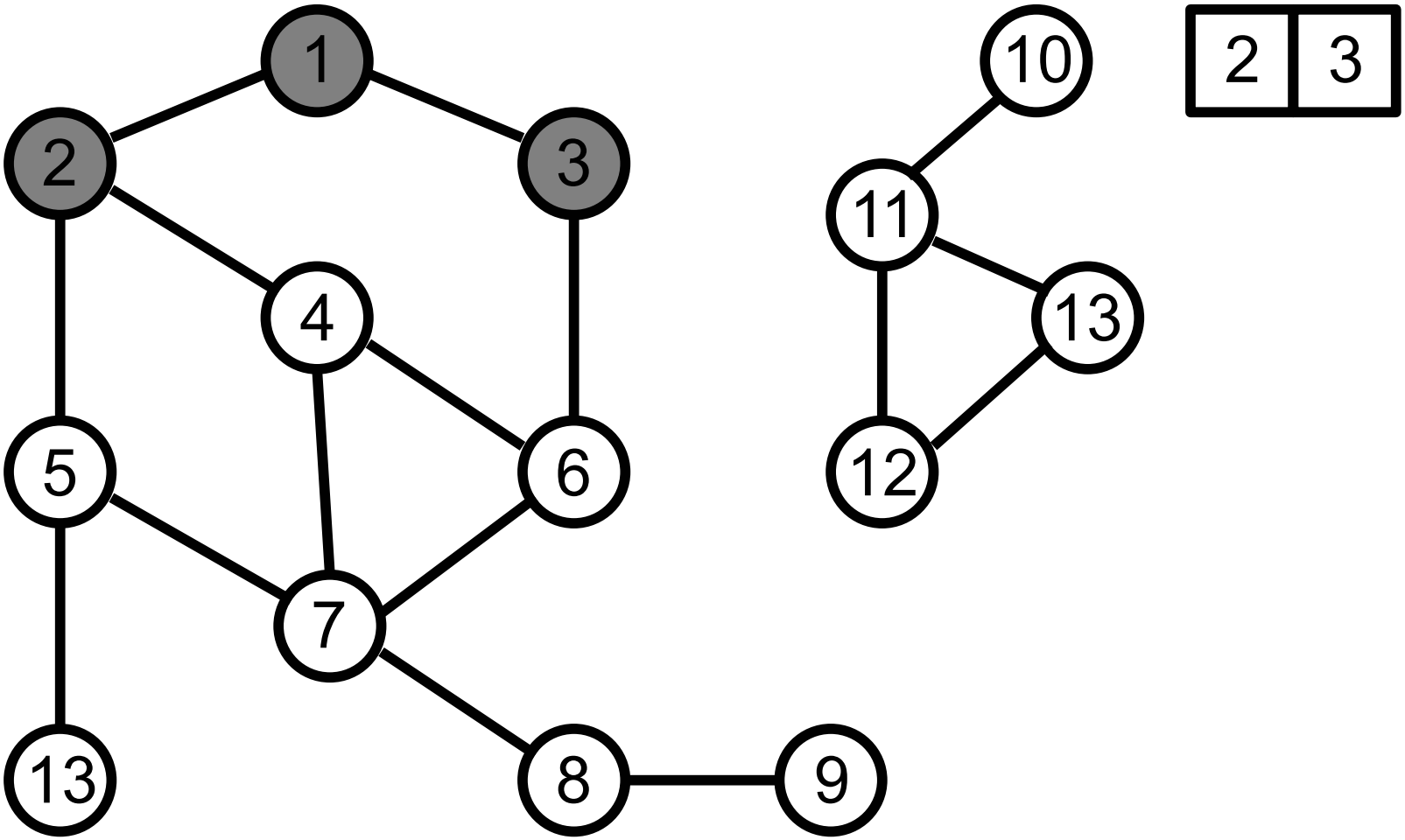
Przeglądanie w głąb



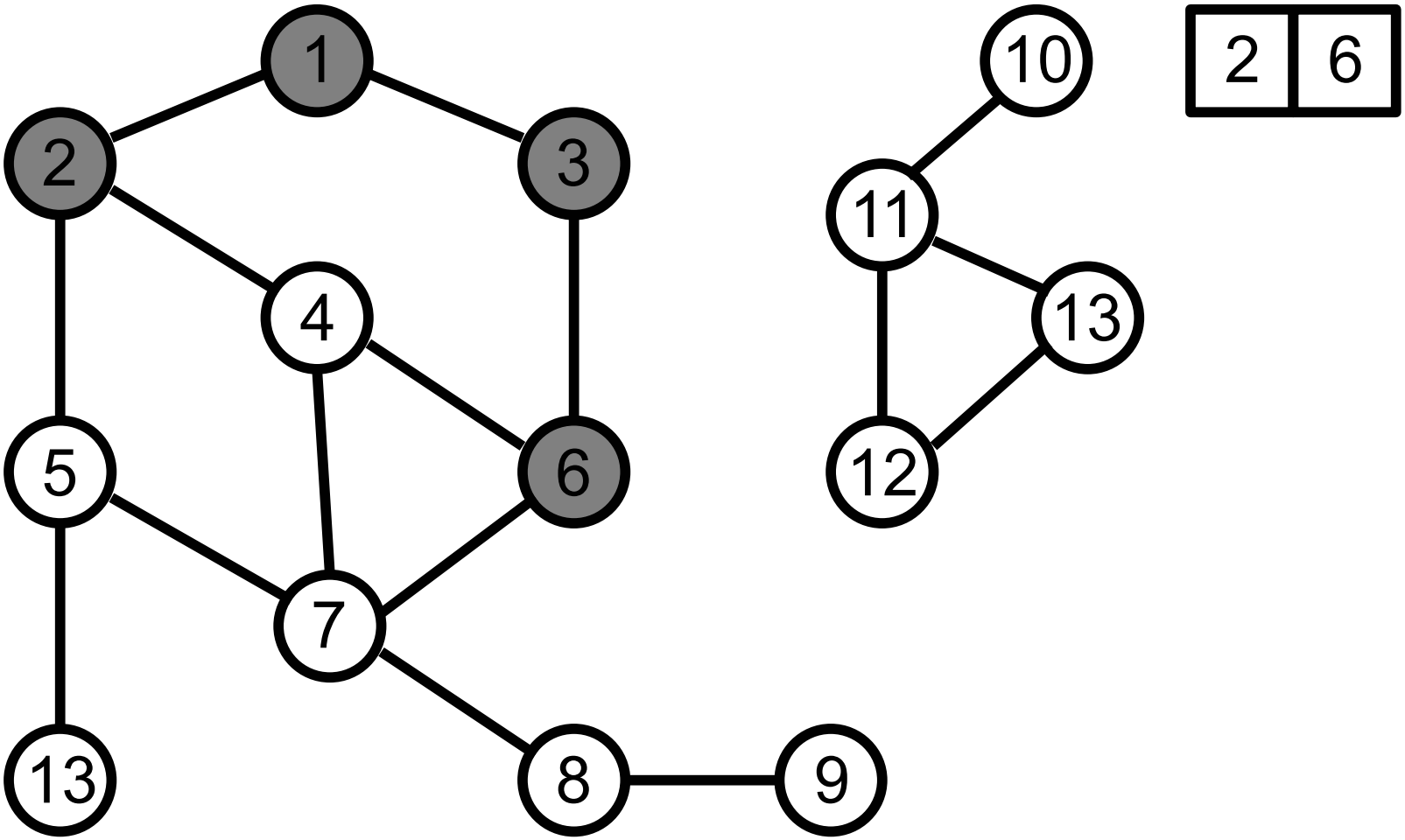
Przeglądanie w głąb



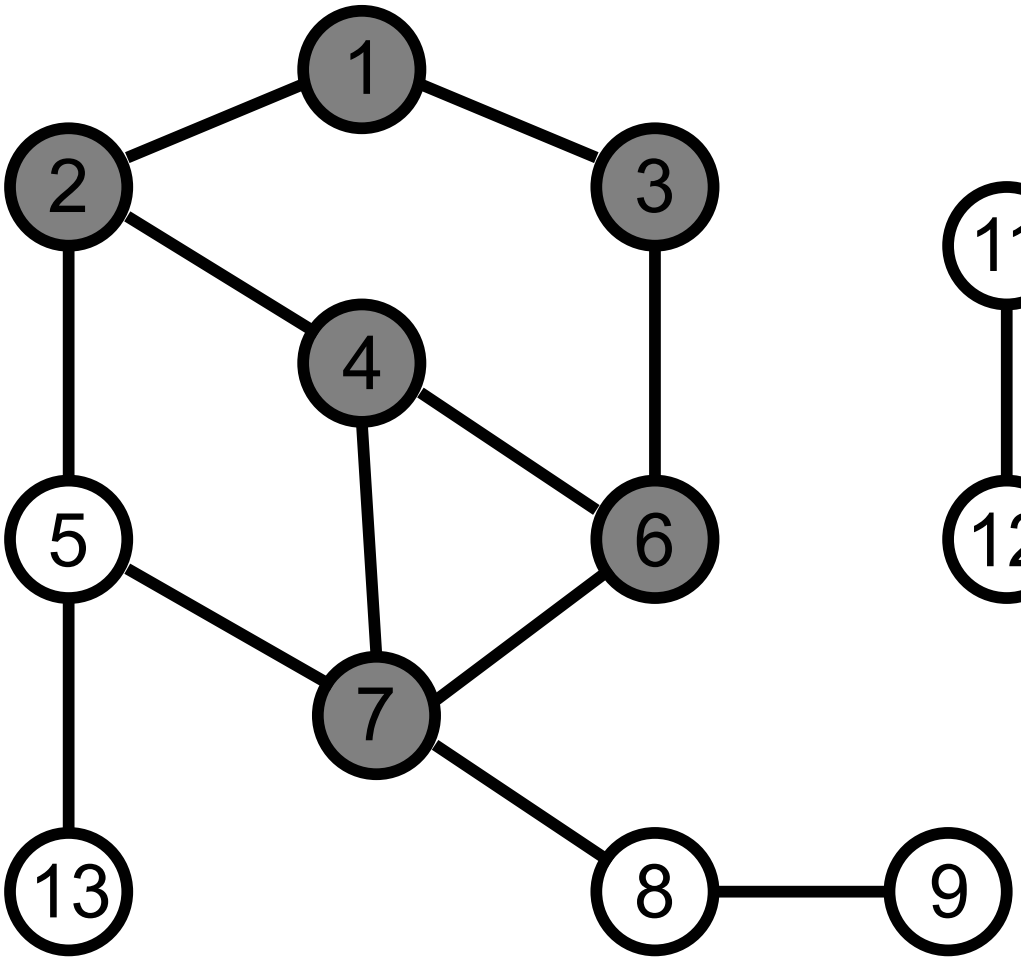
Przeglądanie w głąb

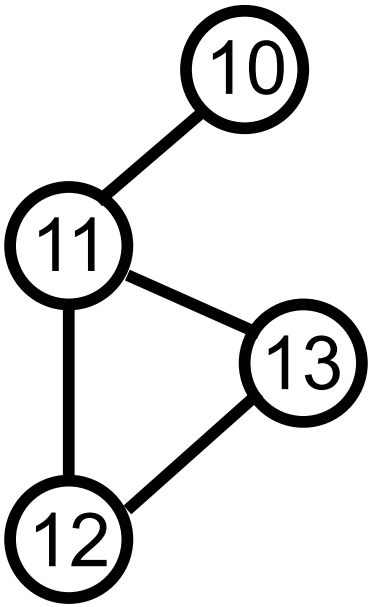


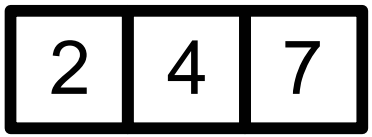
Przeglądanie w głąb



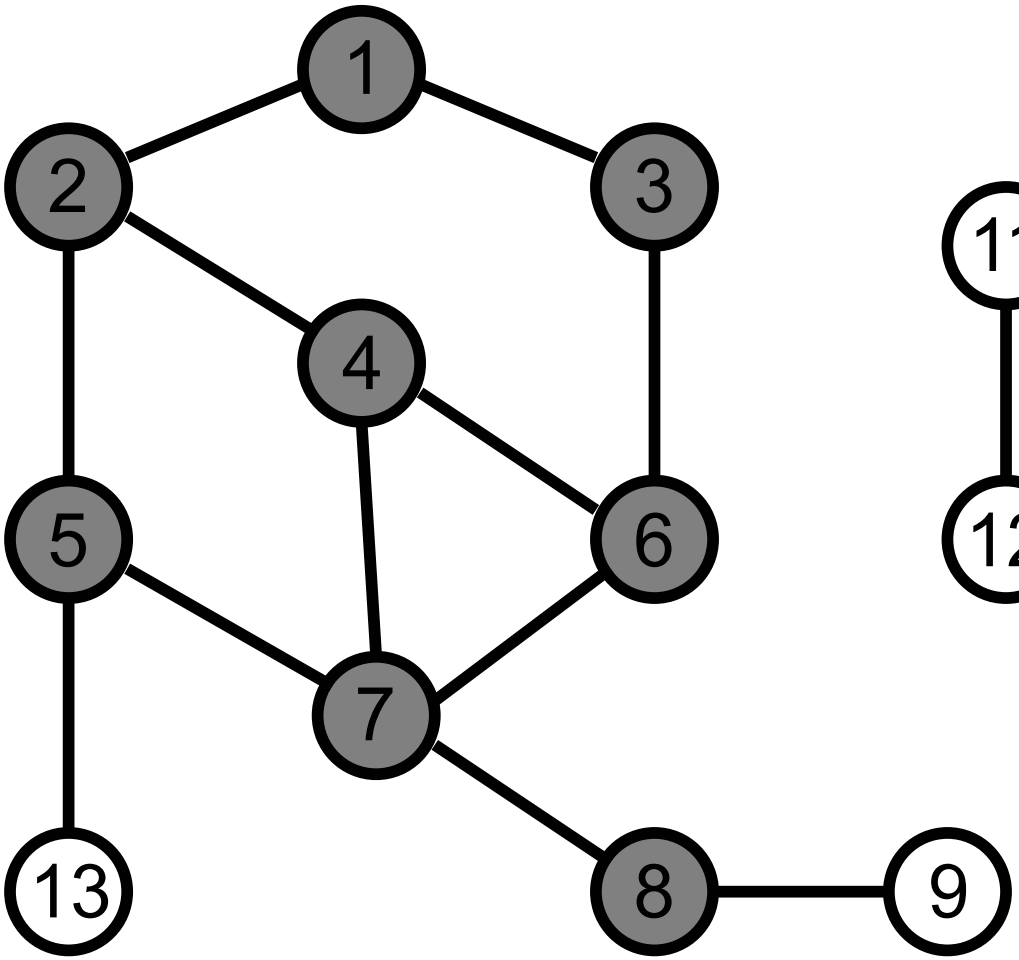
Przeglądanie w głąb

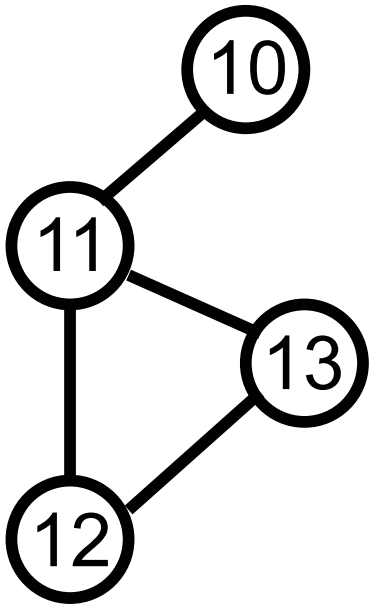






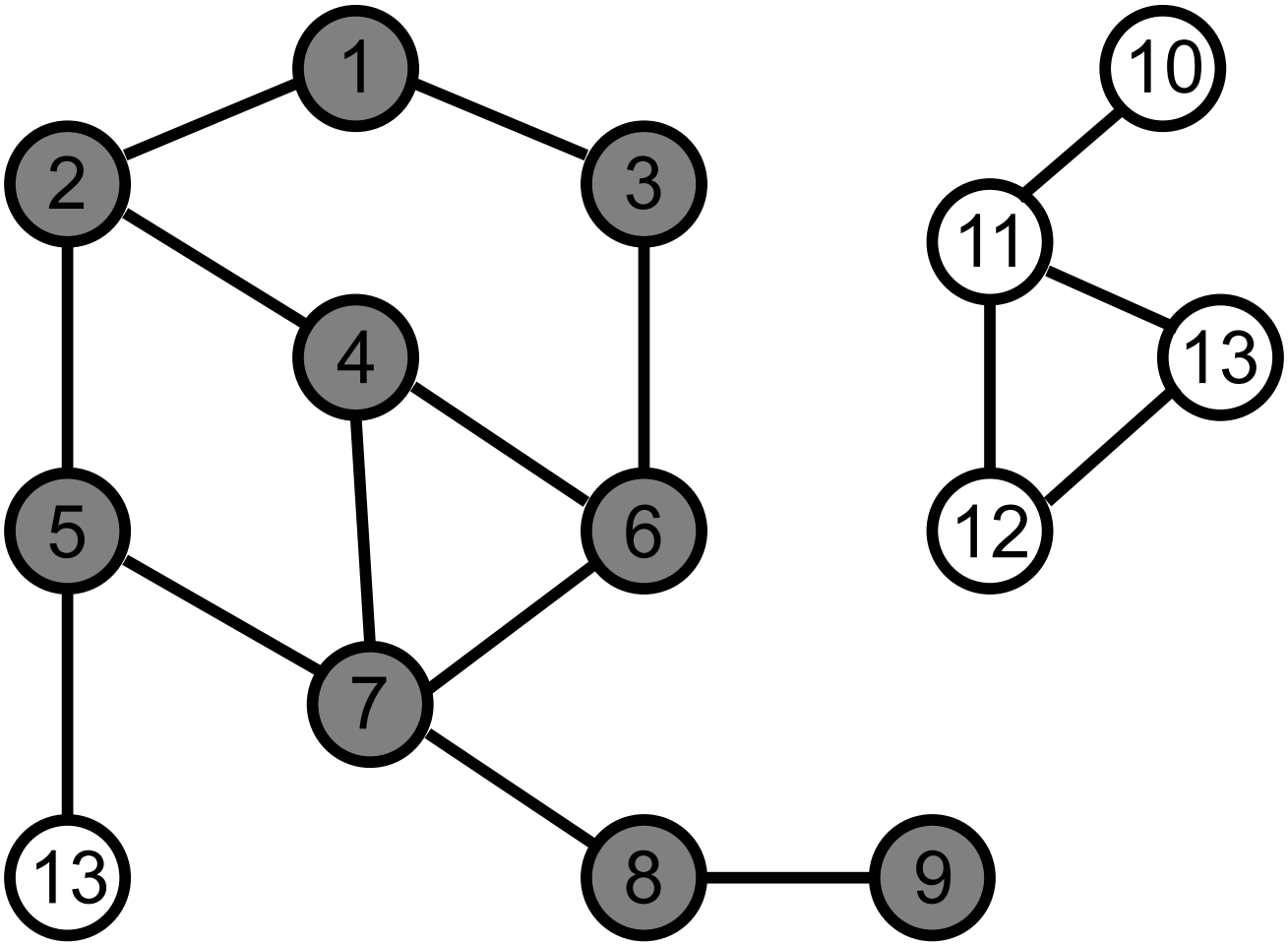
Przeglądanie w głąb





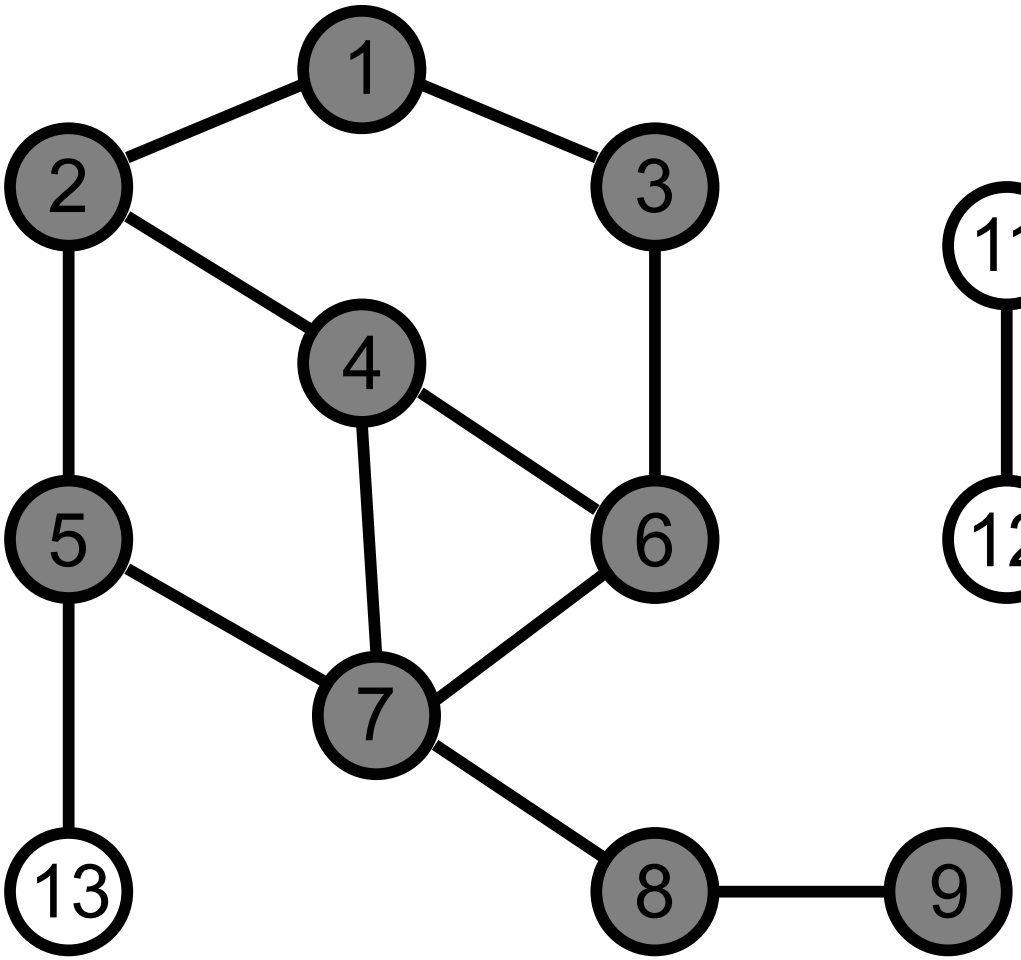
2	4	5	8

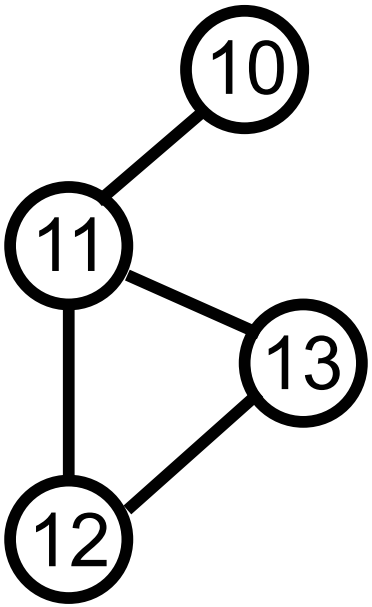
Przeglądanie w głąb



2	4	5	9

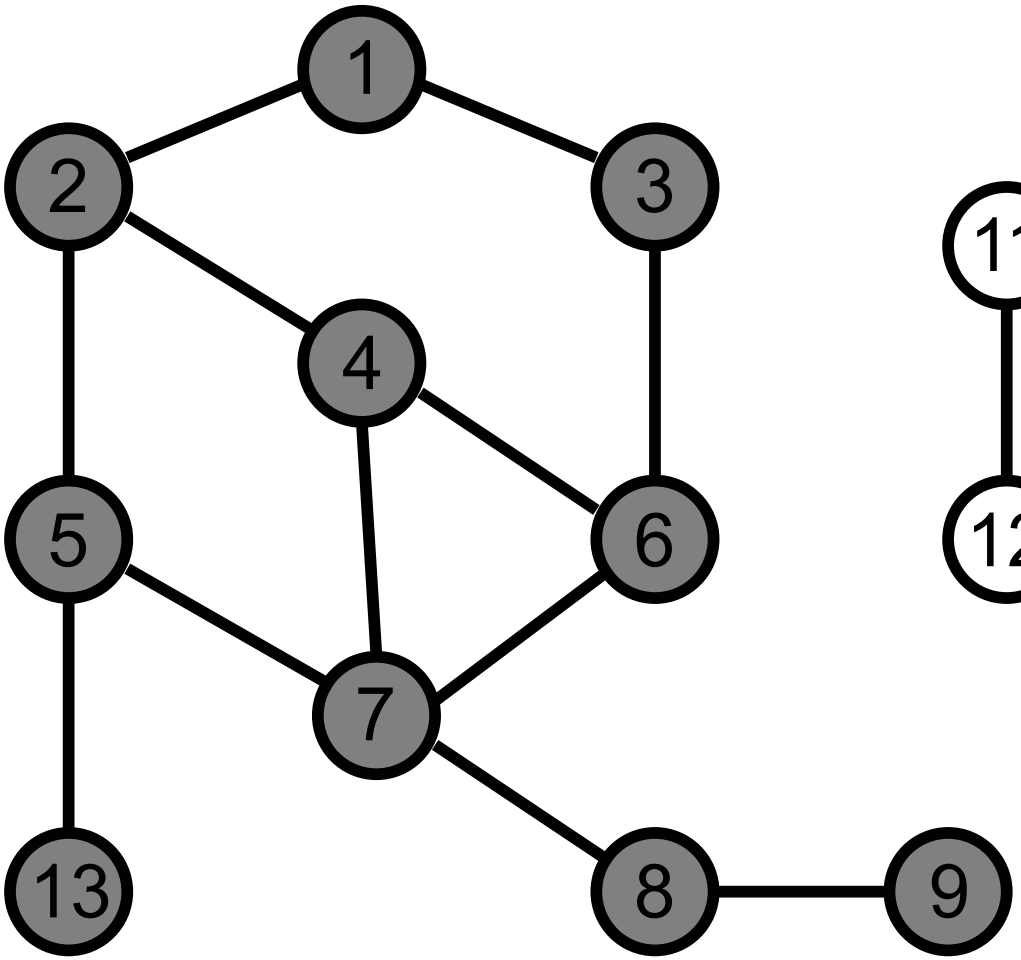
Przeglądanie w głąb

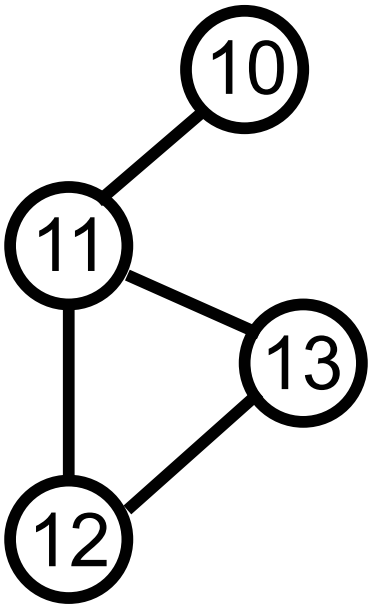




2	4	5

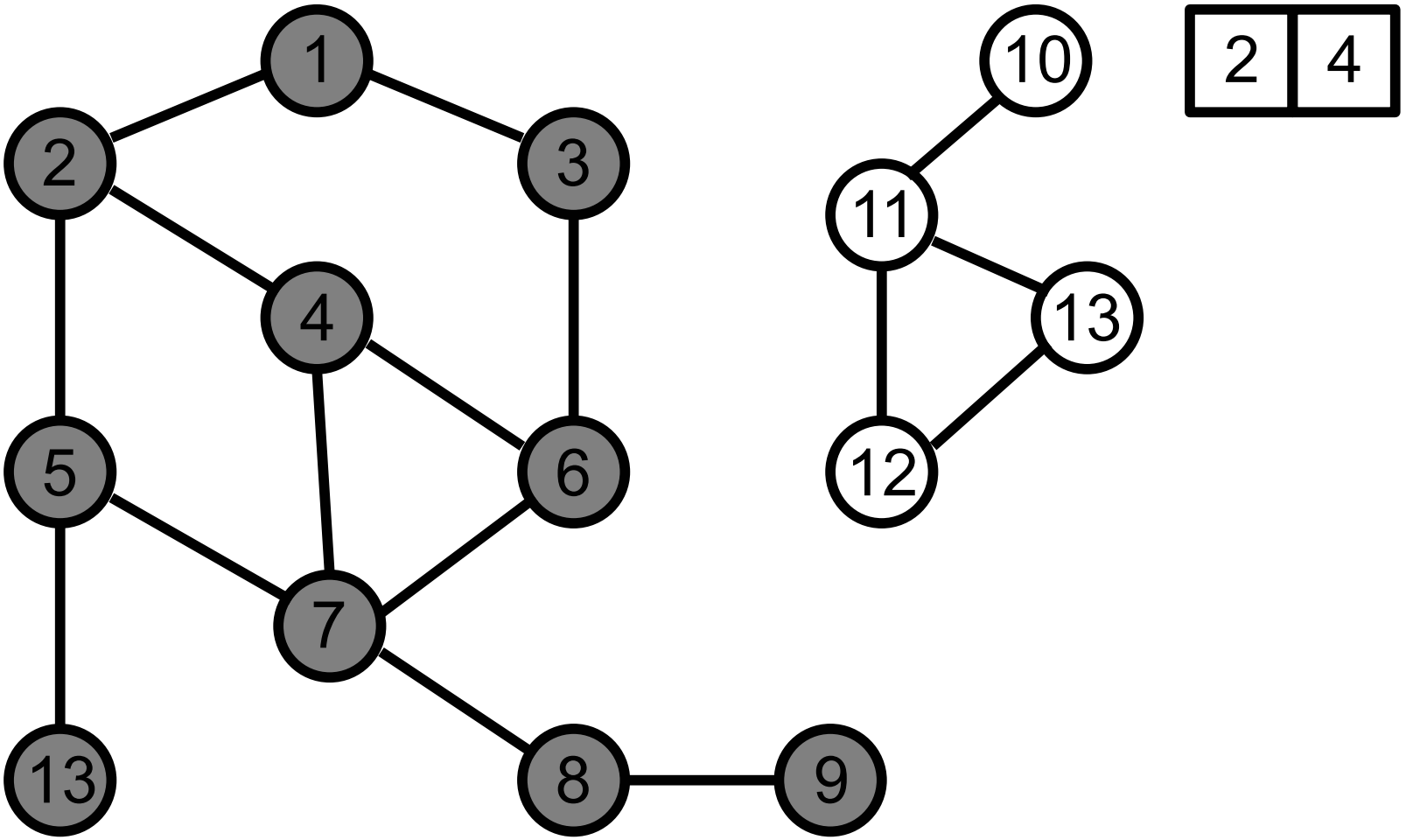
Przeglądanie w głąb



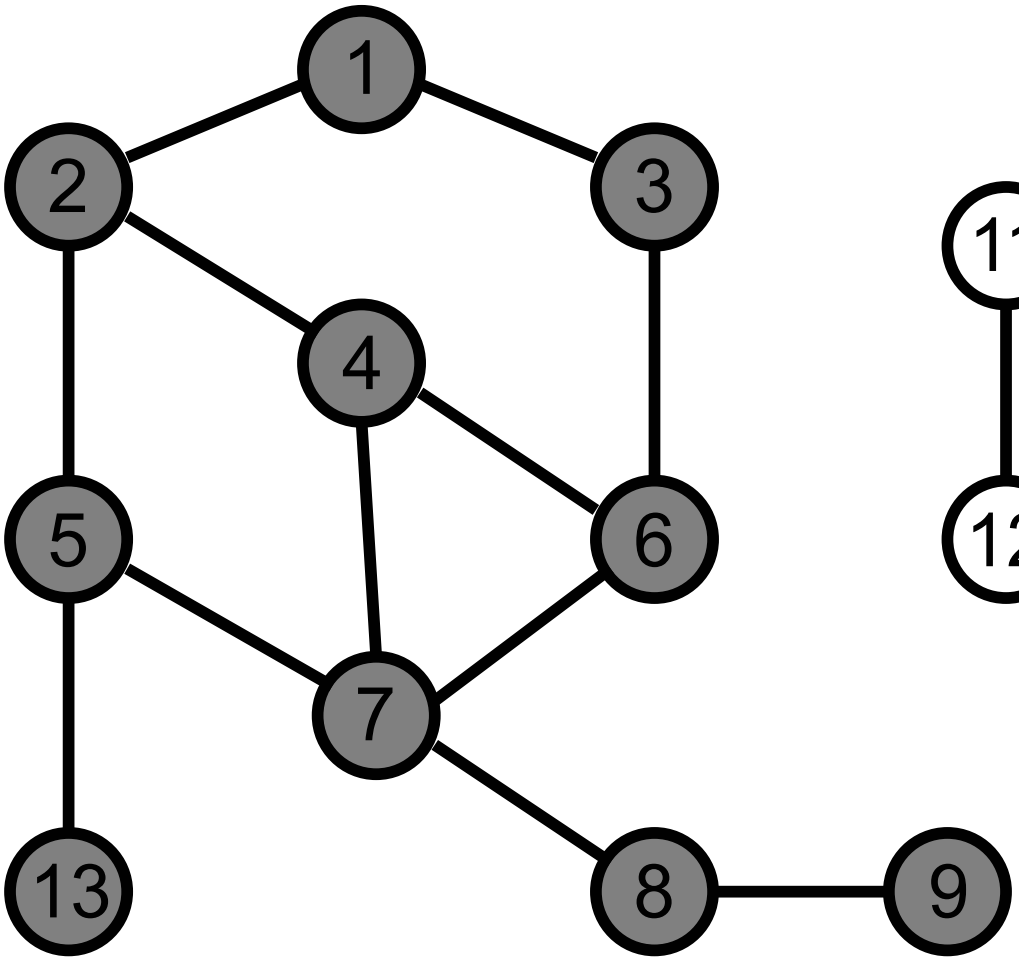


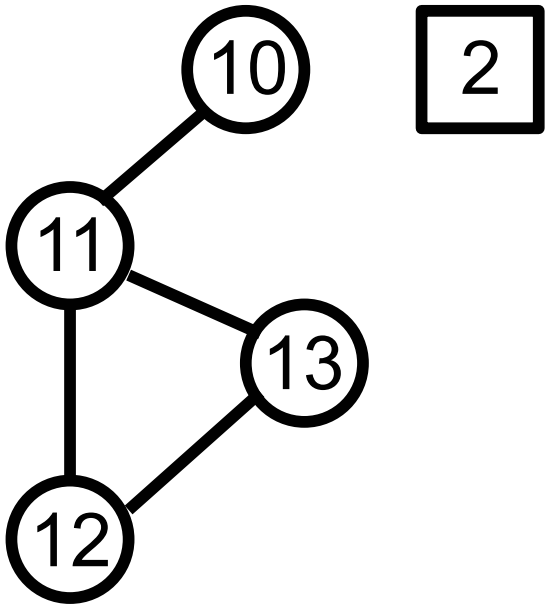
2	4	13

Przeglądanie w głąb

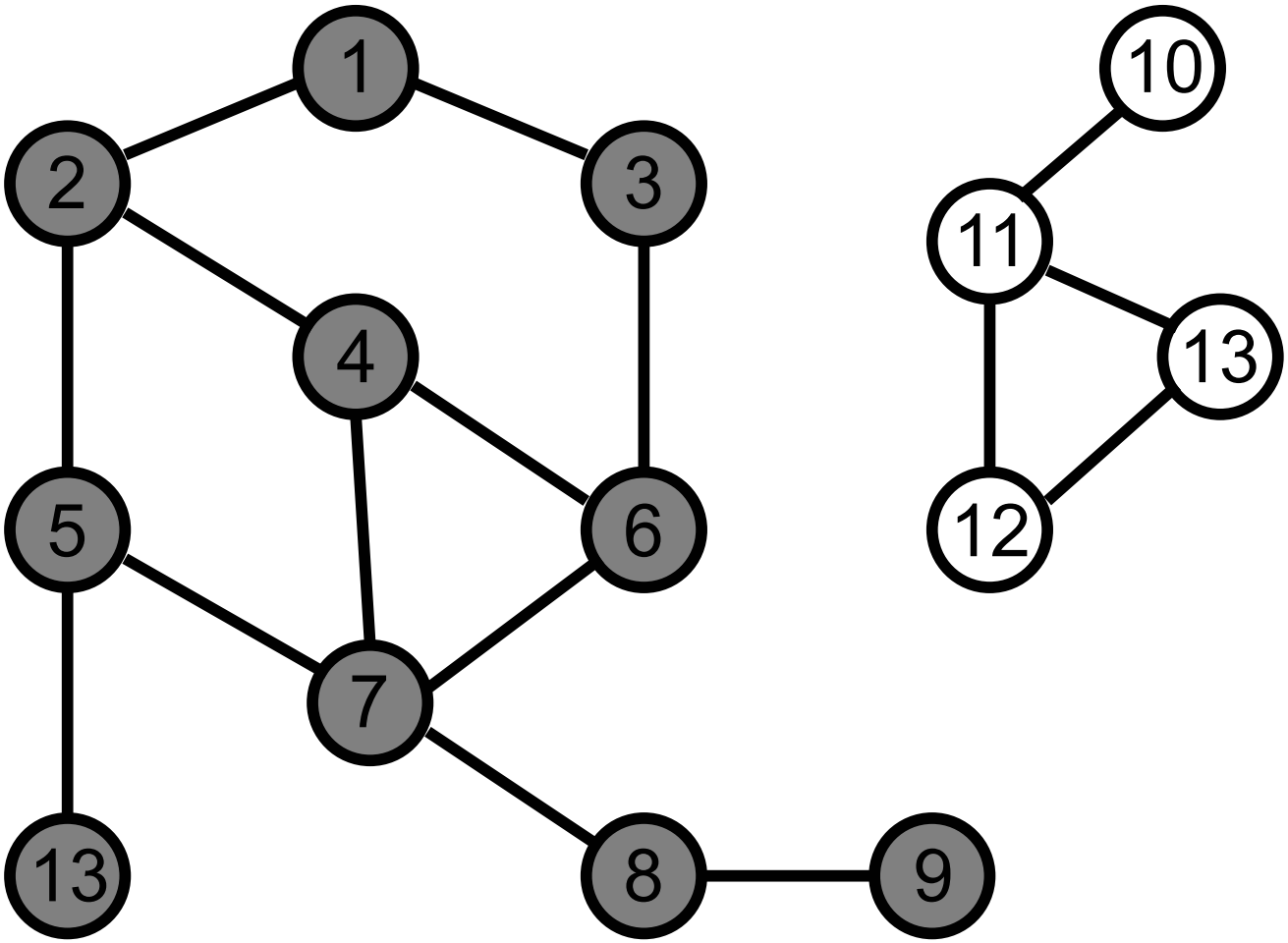


Przeglądanie w głąb





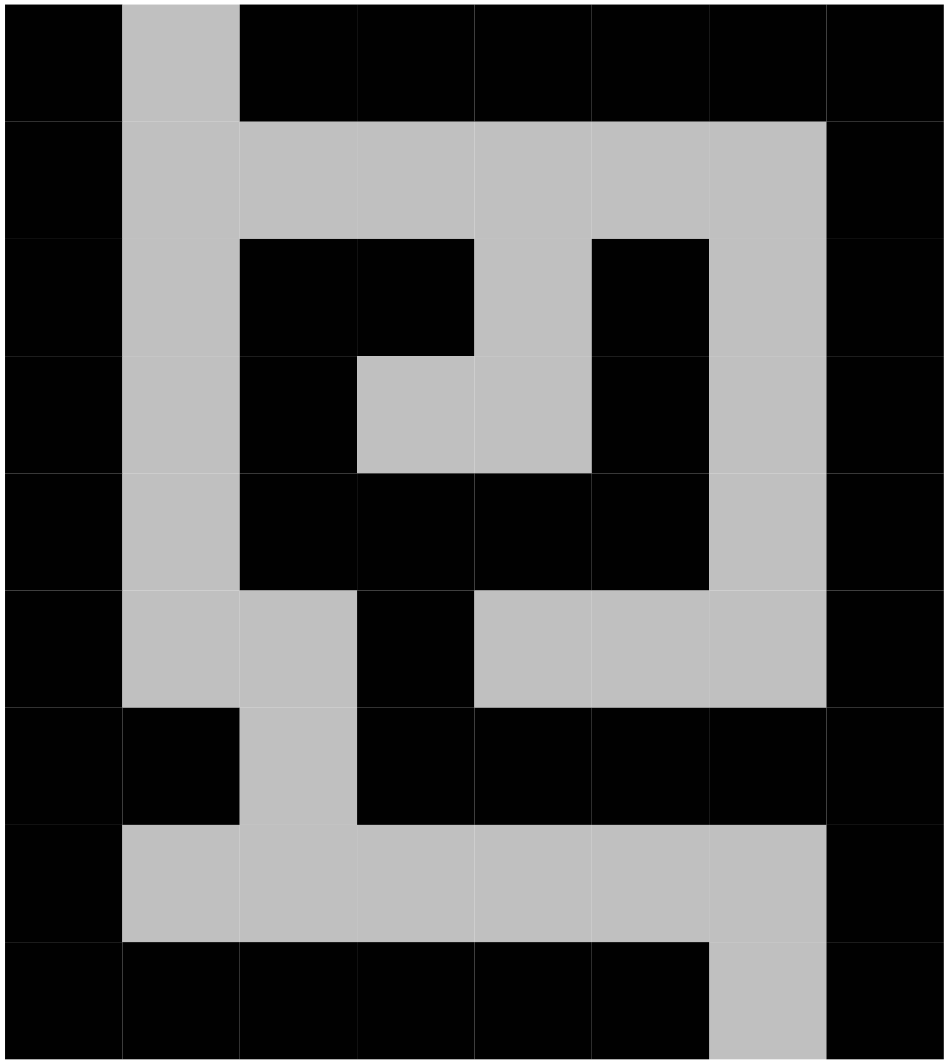
Przeglądanie w głąb



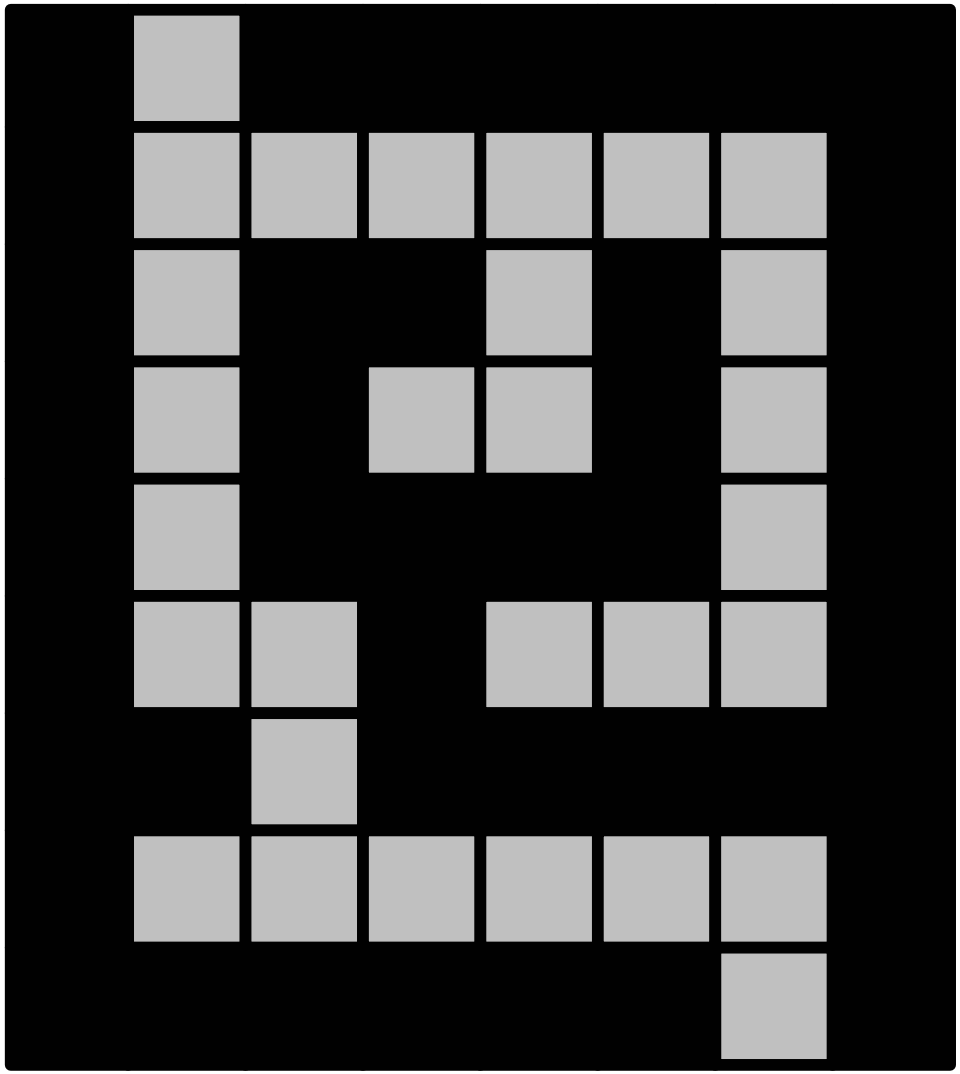
Przeglądanie w głąb - iteracja (uproszczona)

$stos =$ pusty stos
for $i = 1, \ldots, n$ **do** $odw[i] = False$
$stos.push(v_s)$
$odw[v_s] = True$
while not $stos.empty()$ **do**
    $v = stos.top()$
    $stos.pop()$
    for $u =$ każdy sąsiad v **do**
        if $odw[u]$ **then continue**
        $stos.push(u)$
        $odw[u] = True$
    end for
end while

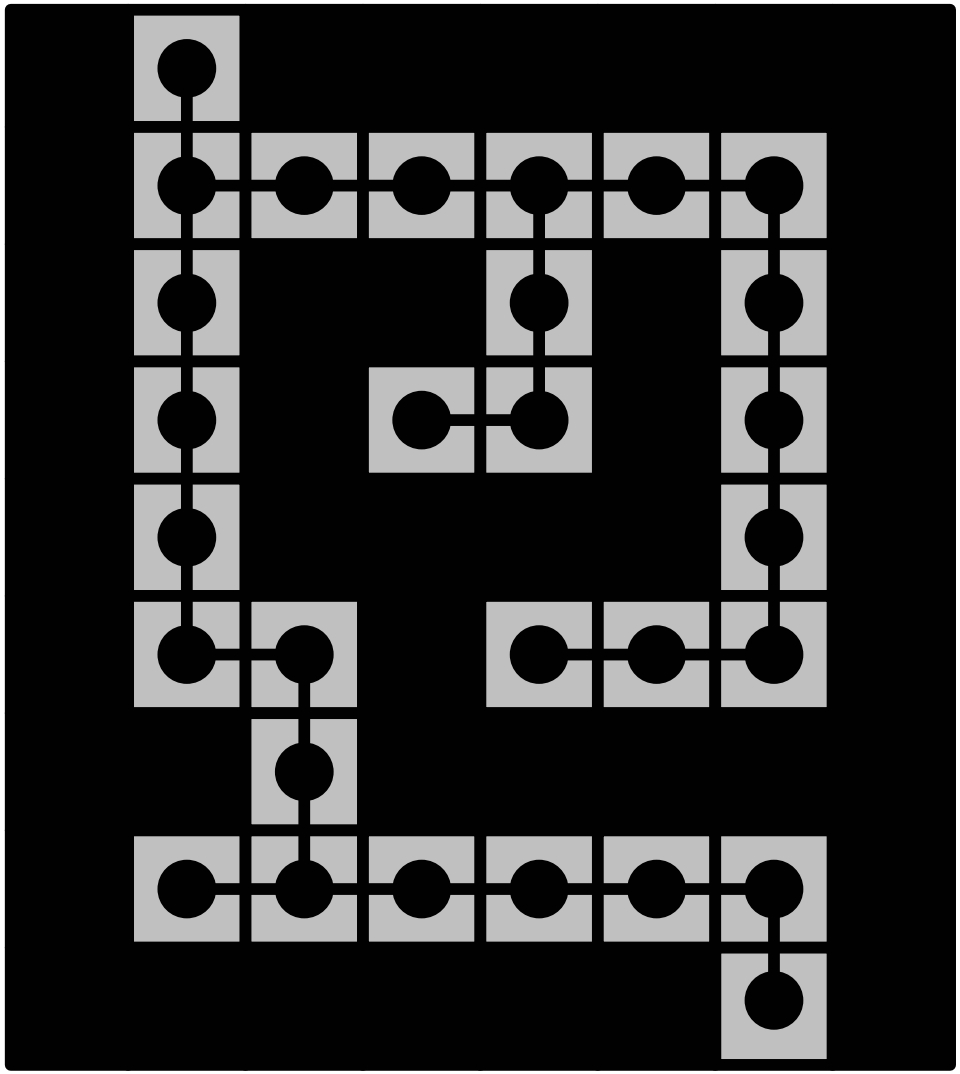
Znajdowanie drogi w labiryncie



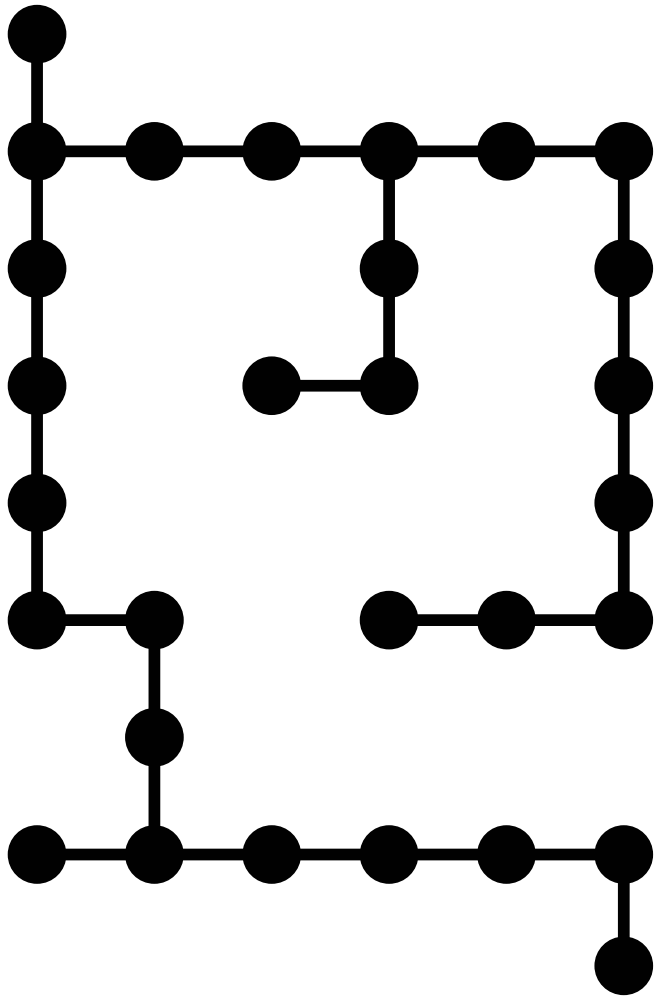
Znajdowanie drogi w labiryncie



Znajdowanie drogi w labiryncie



Znajdowanie drogi w labiryncie



Znajdowanie drogi w labiryncie

- dla każdego wierzchołka dodatkowo pamiętamy, z którego wierzchołka do niego doszliśmy
- drogę odtwarzamy idąc od końca

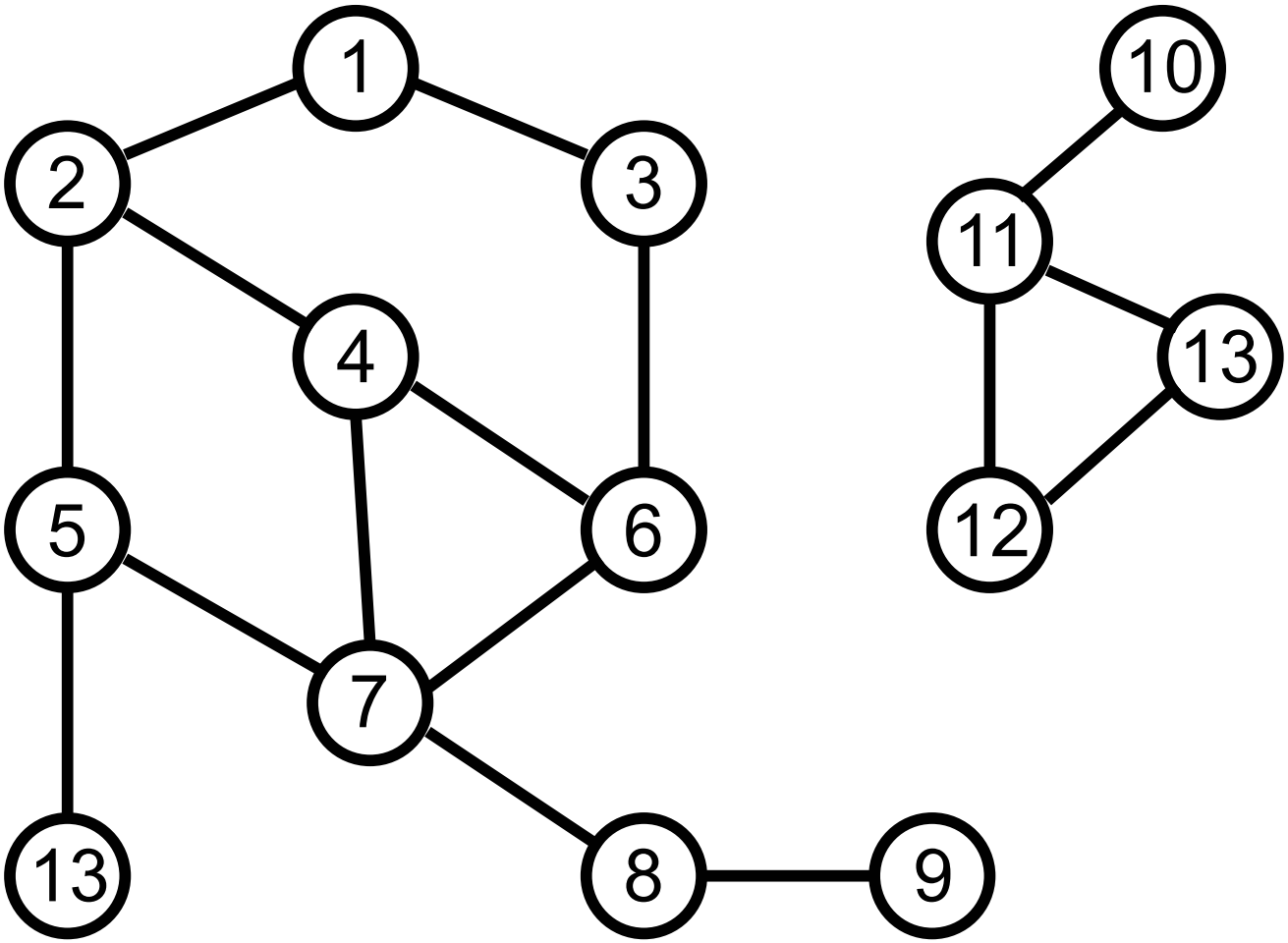
Przeglądanie wszerz

- algorytm odwiedzania wszystkich wierzchołków z jednej spójnej składowej
- przeglądamy graf "warstwami": najpierw wierzchołek startowy, później jego sąsiadów, sąsiadów sąsiadów, itp.
- implementacja: stos zastępujemy kolejką
- trudniejsze w implementacji (brak prostej rekurencji) ale daje więcej informacji – do wierzchołka dochodzimy najkrótszą ścieżką z wierzchołka startowego
- np. wyszukiwanie najkrótszej drogi w labiryncie

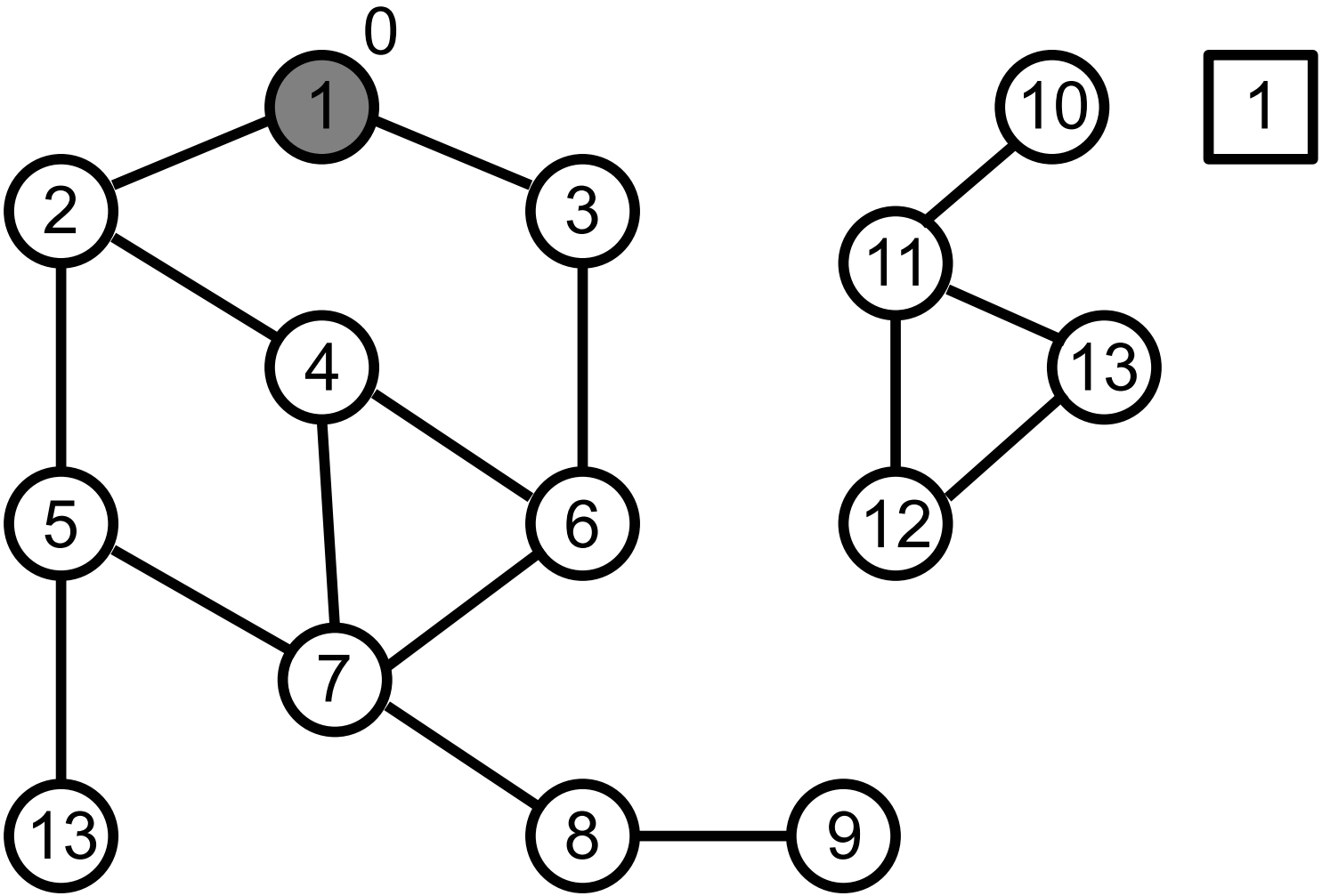
Przeglądanie wszerz

kolejka = pusta kolejka
for $i = 1, \ldots, n$ **do** $odw[i] = False$
$kolejka.enqueue(v_s)$
$odw[v_s] = True$
while Q nie jest pusta **do**
    $v = kolejka.front()$
    $kolejka.dequeue()$
    for $u =$ każdy sąsiad v **do**
        if $odw[u]$ **then continue**
        $kolejka.enqueue(u)$
        $odw[u] = True$
    end for
end while

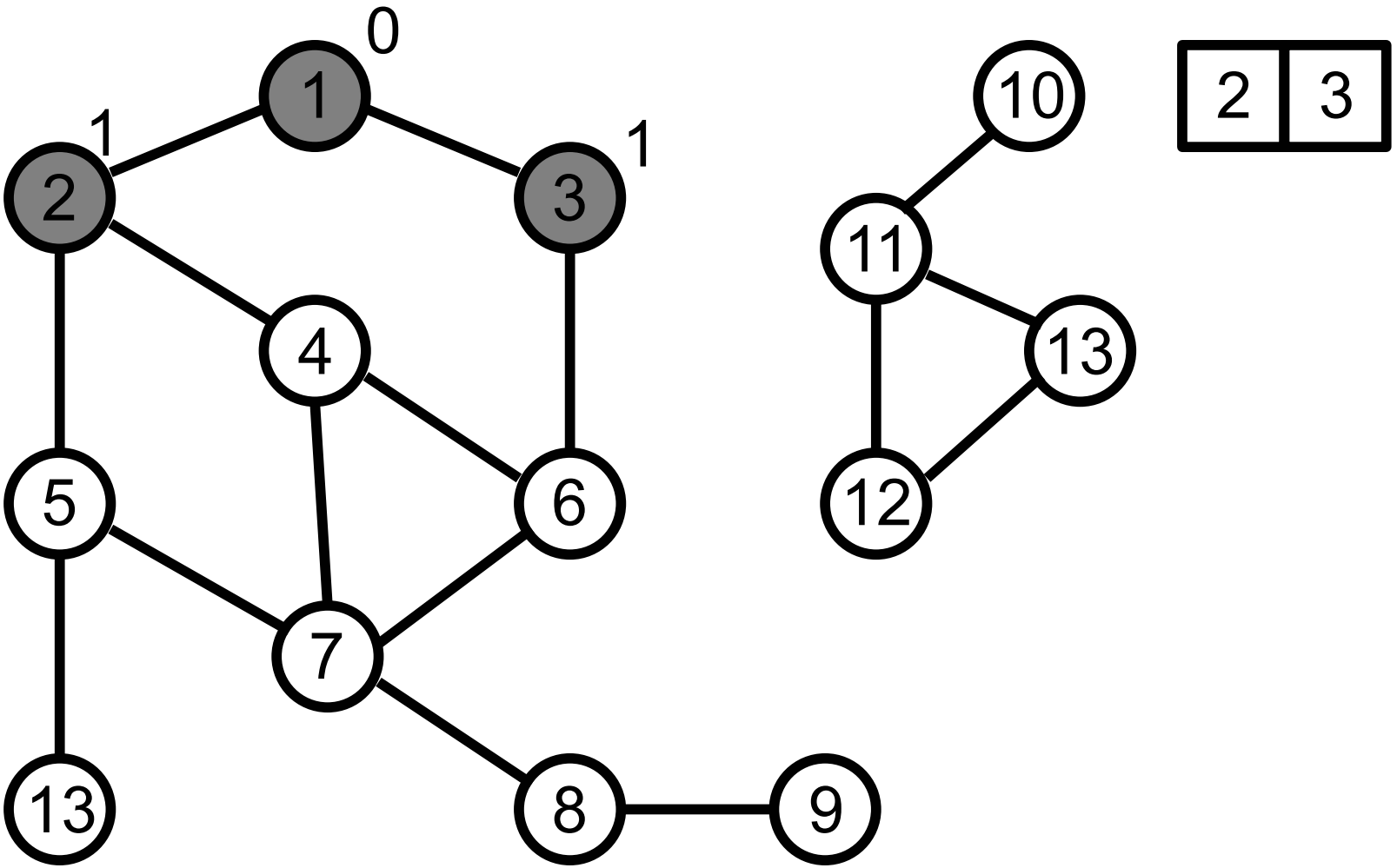
Przeglądanie wszerz



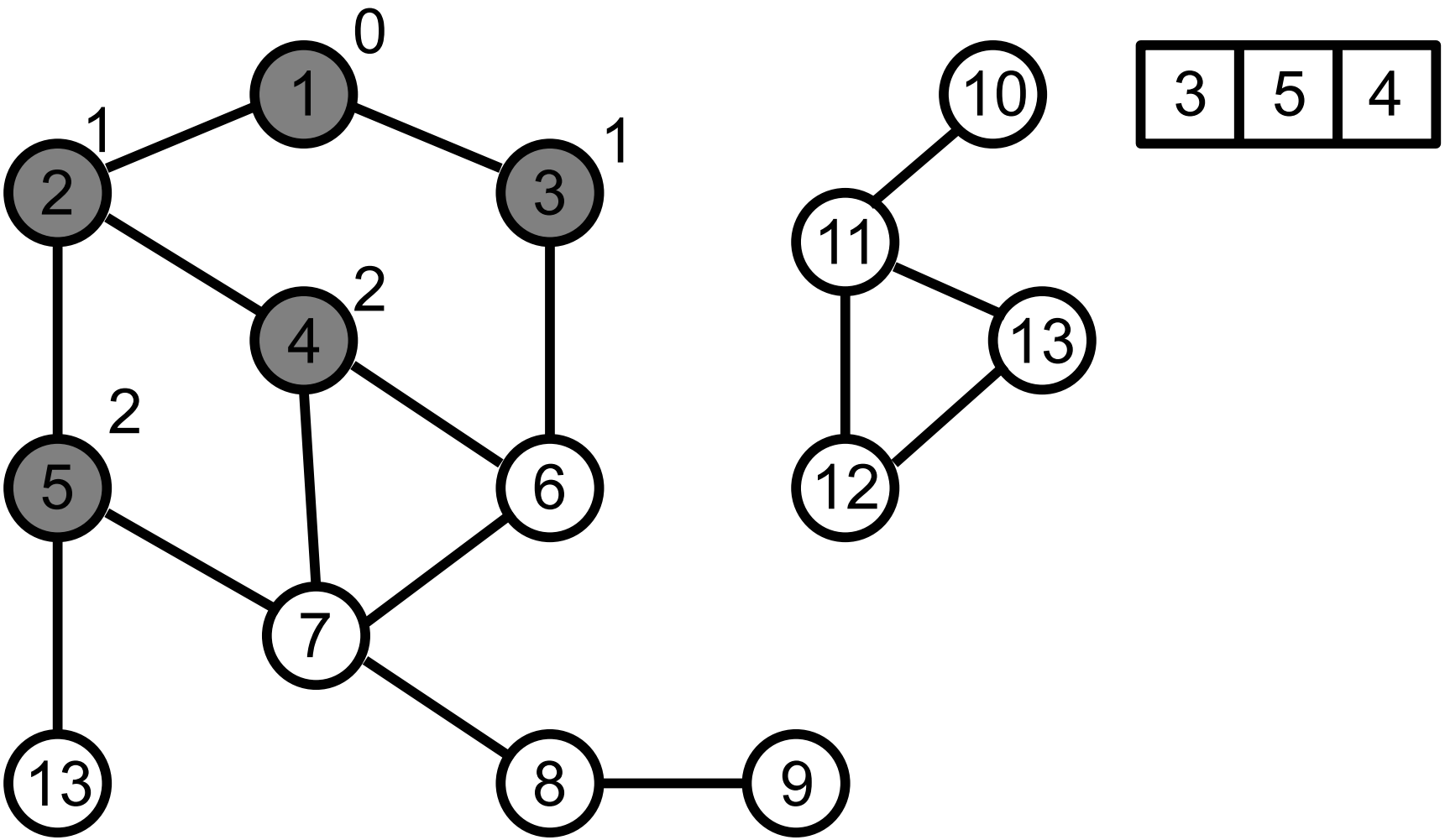
Przeglądanie wszerz



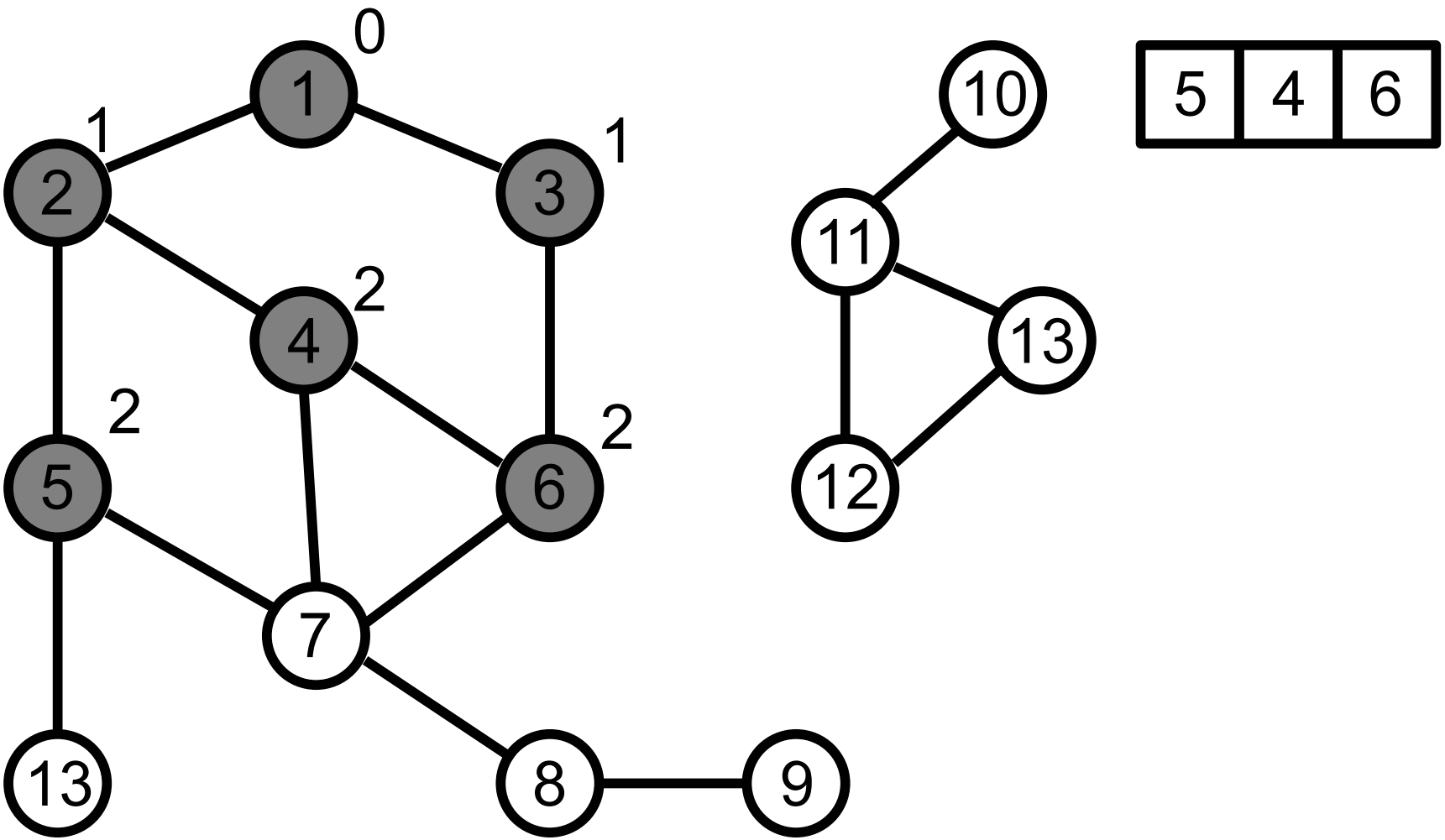
Przeglądanie wszerz



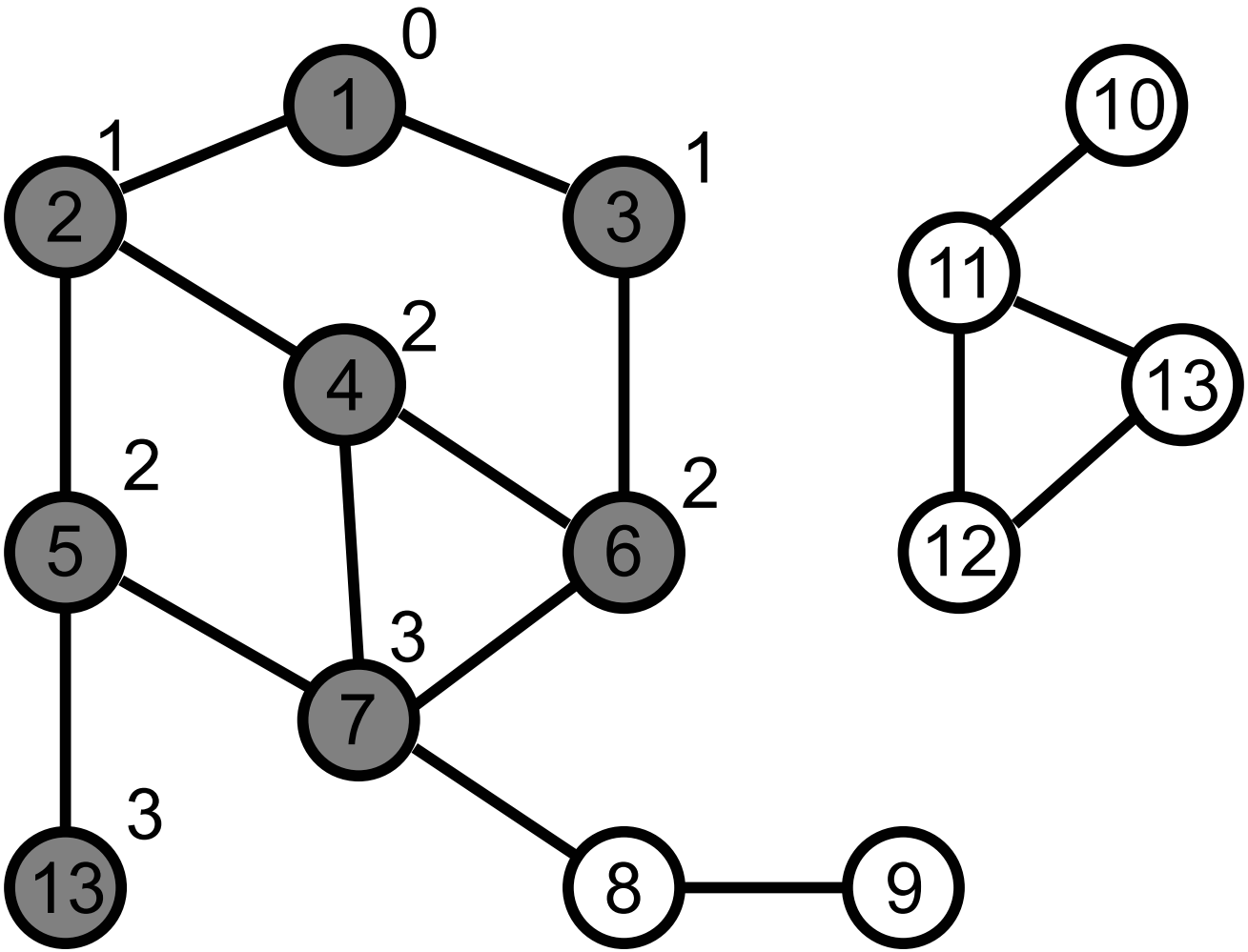
Przeglądanie wszerz



Przeglądanie wszerz

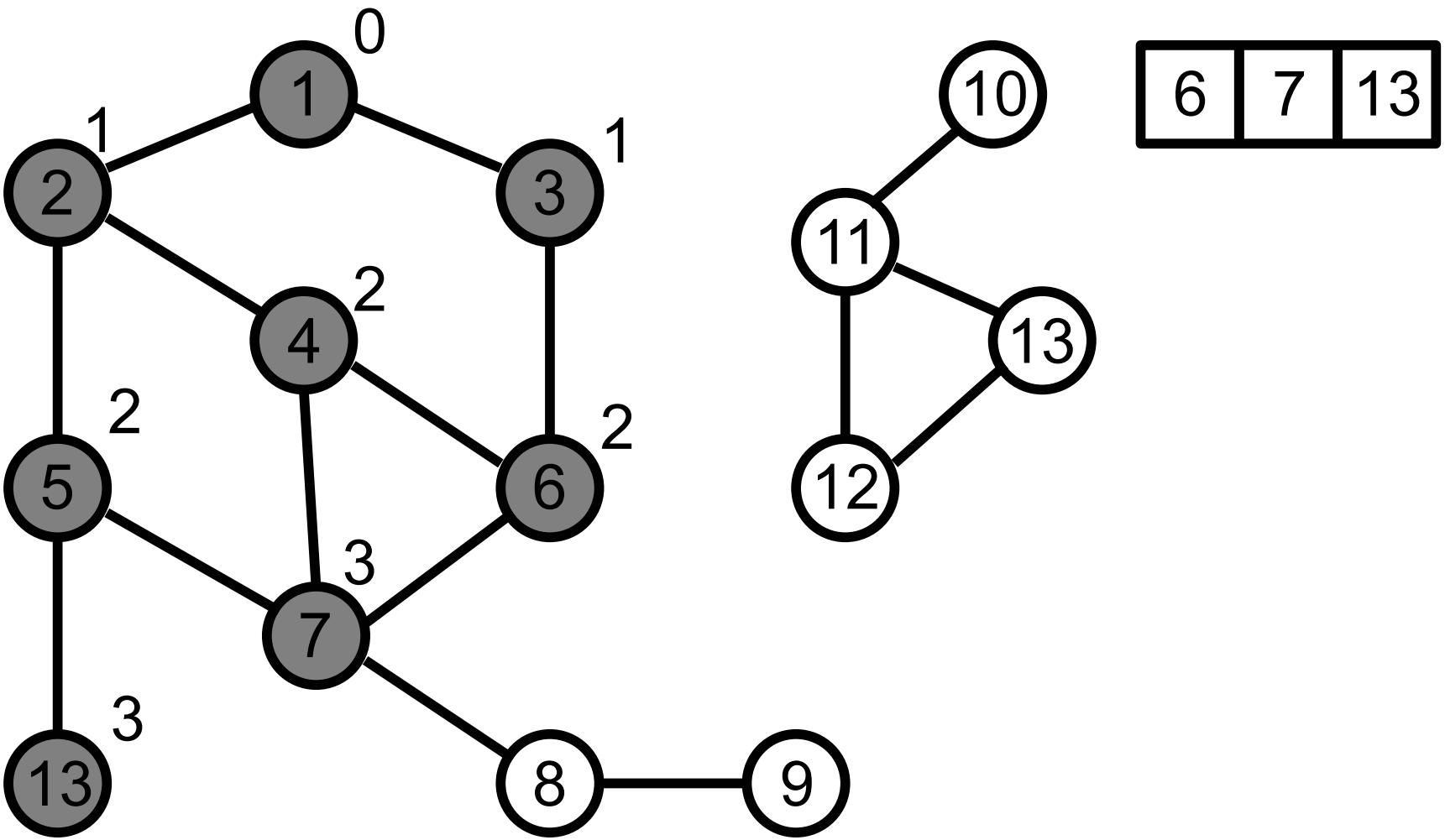


Przeglądanie wszerz

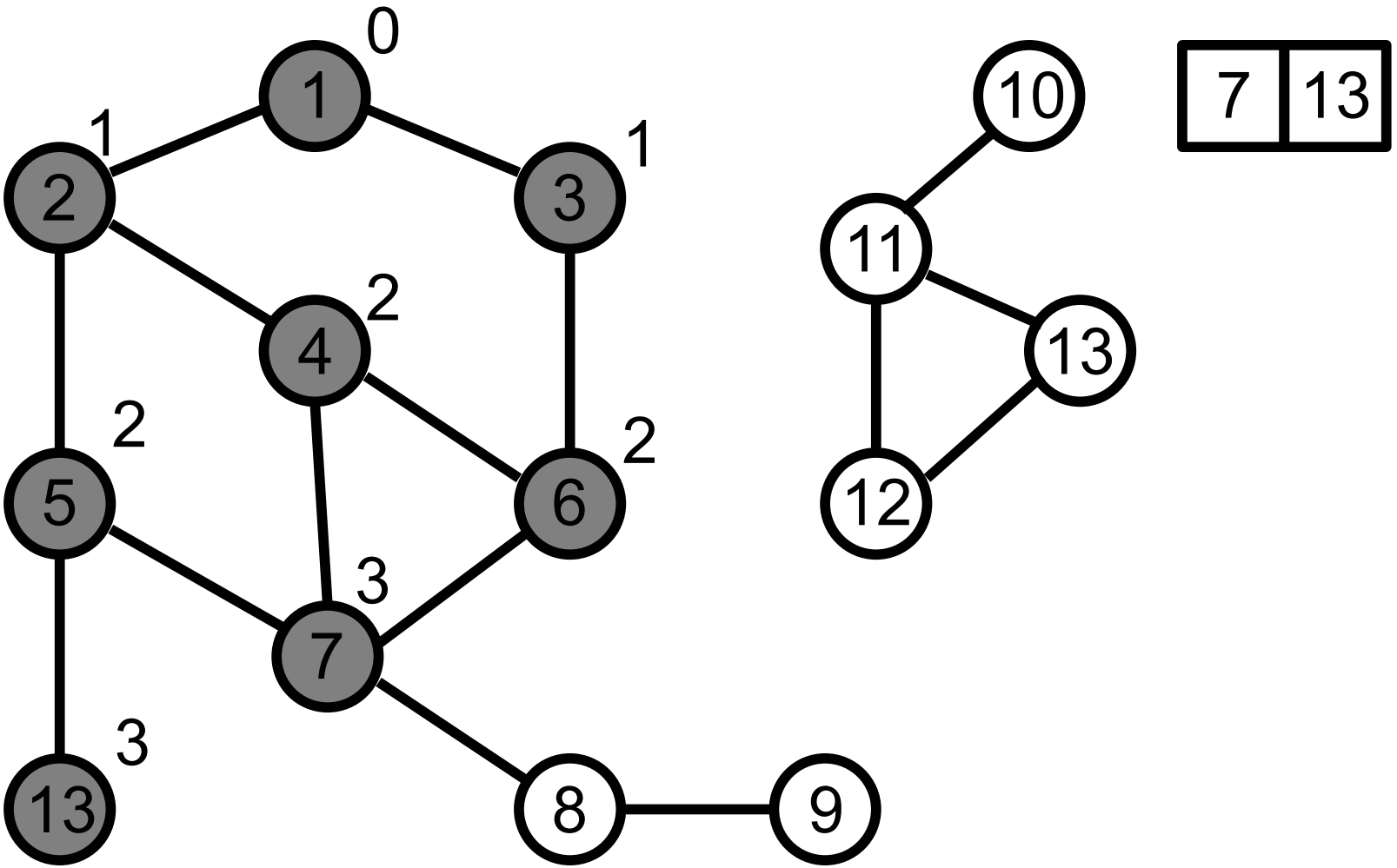


4	6	7	13

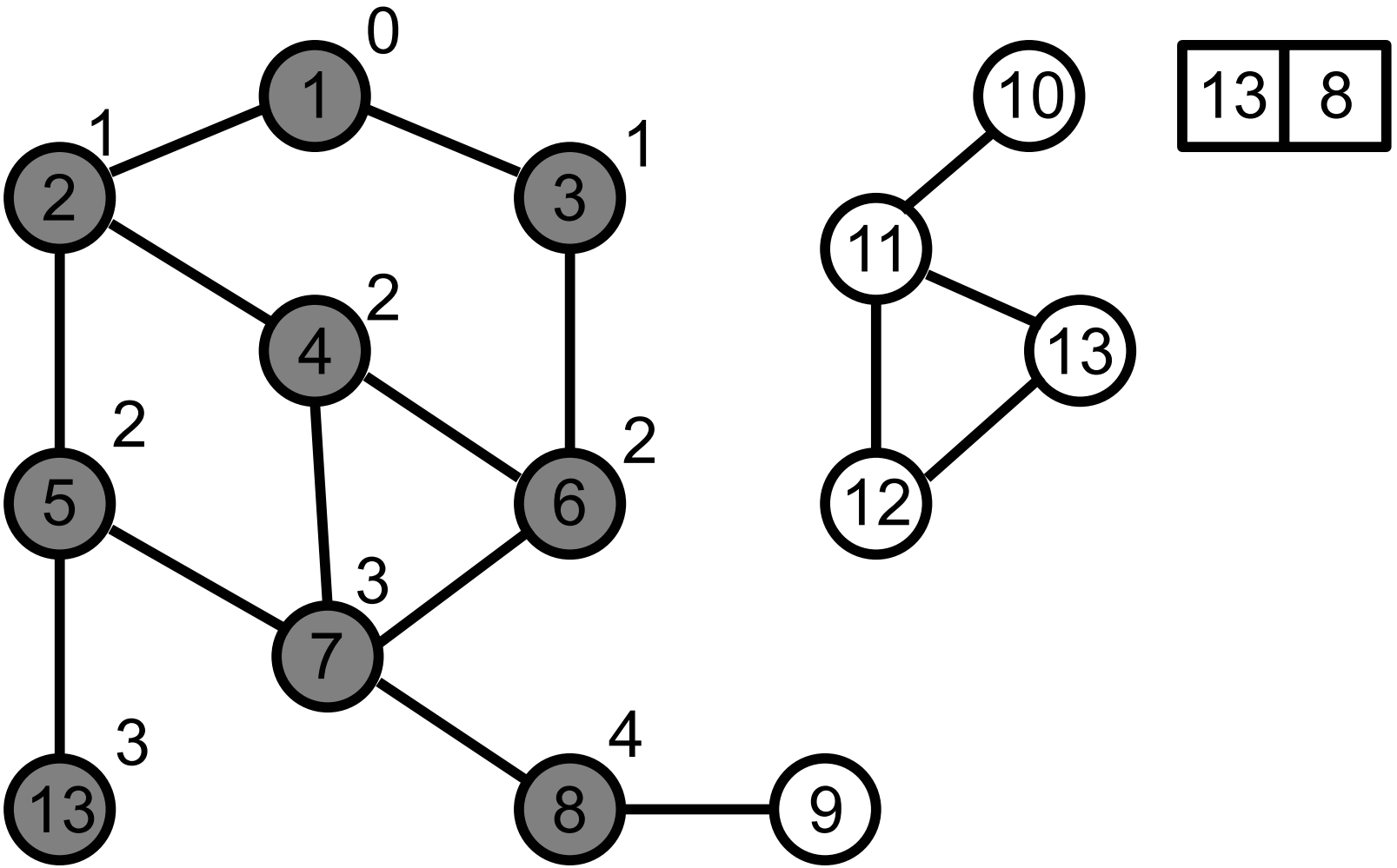
Przeglądanie wszerz



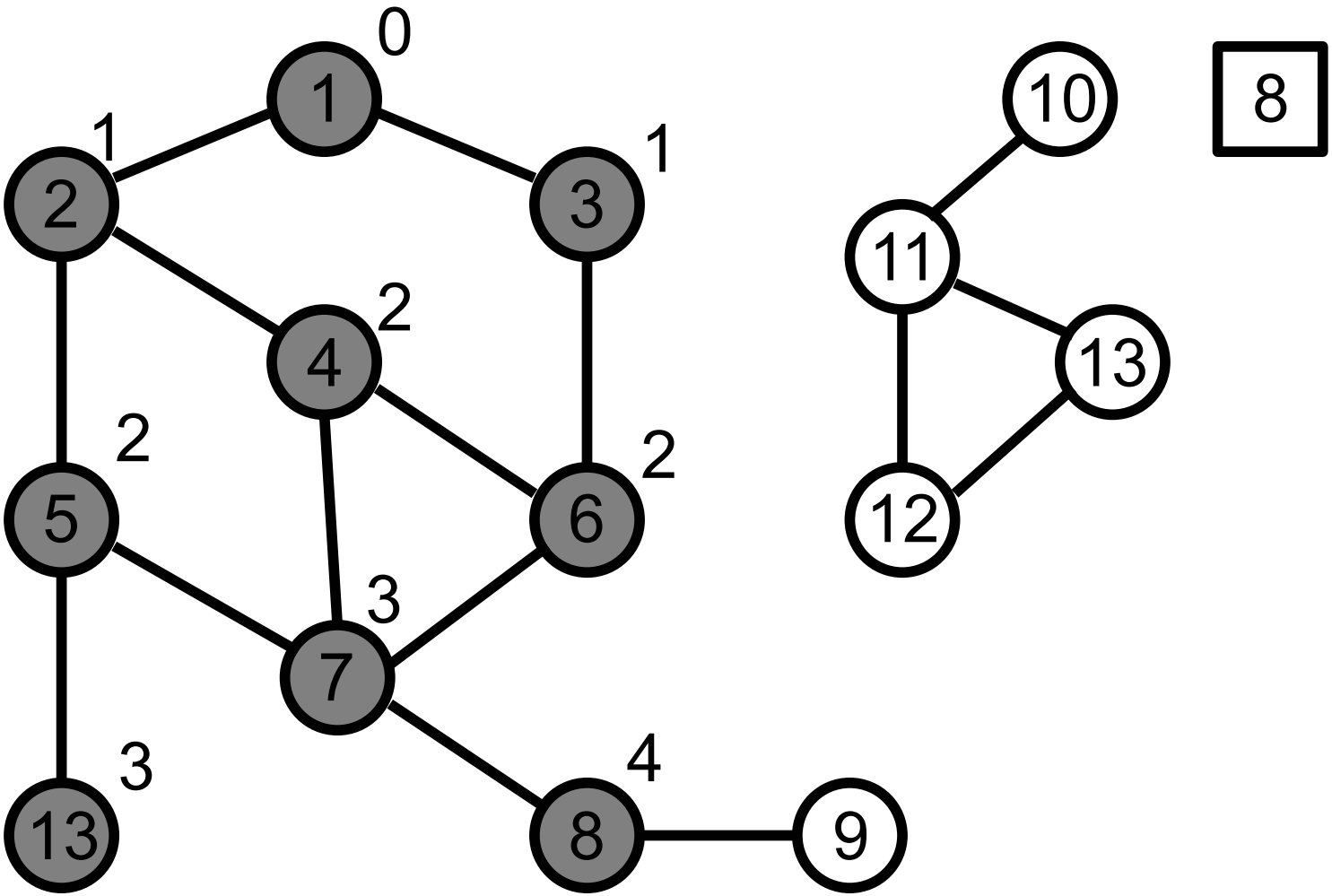
Przeglądanie wszerz



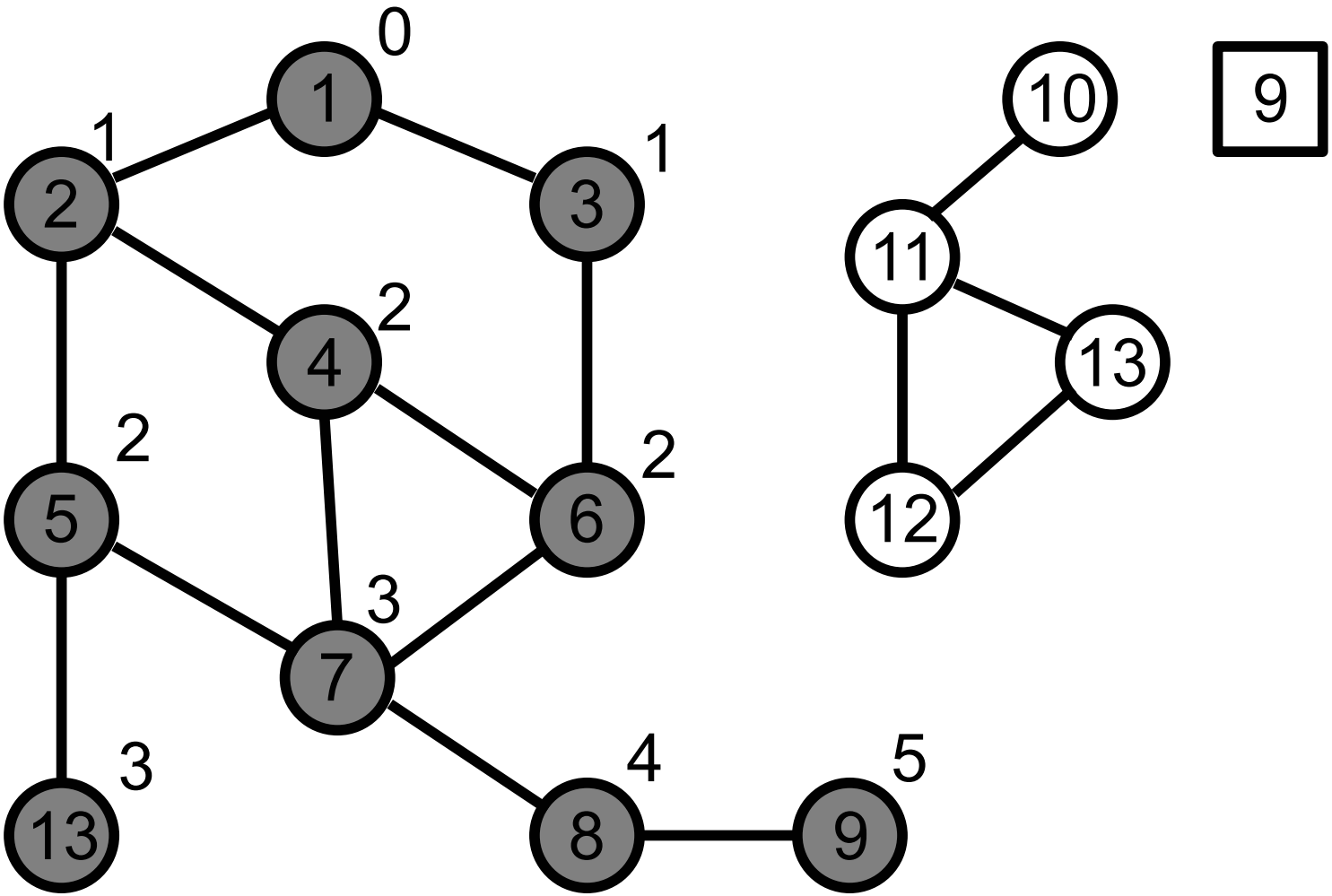
Przeglądanie wszerz



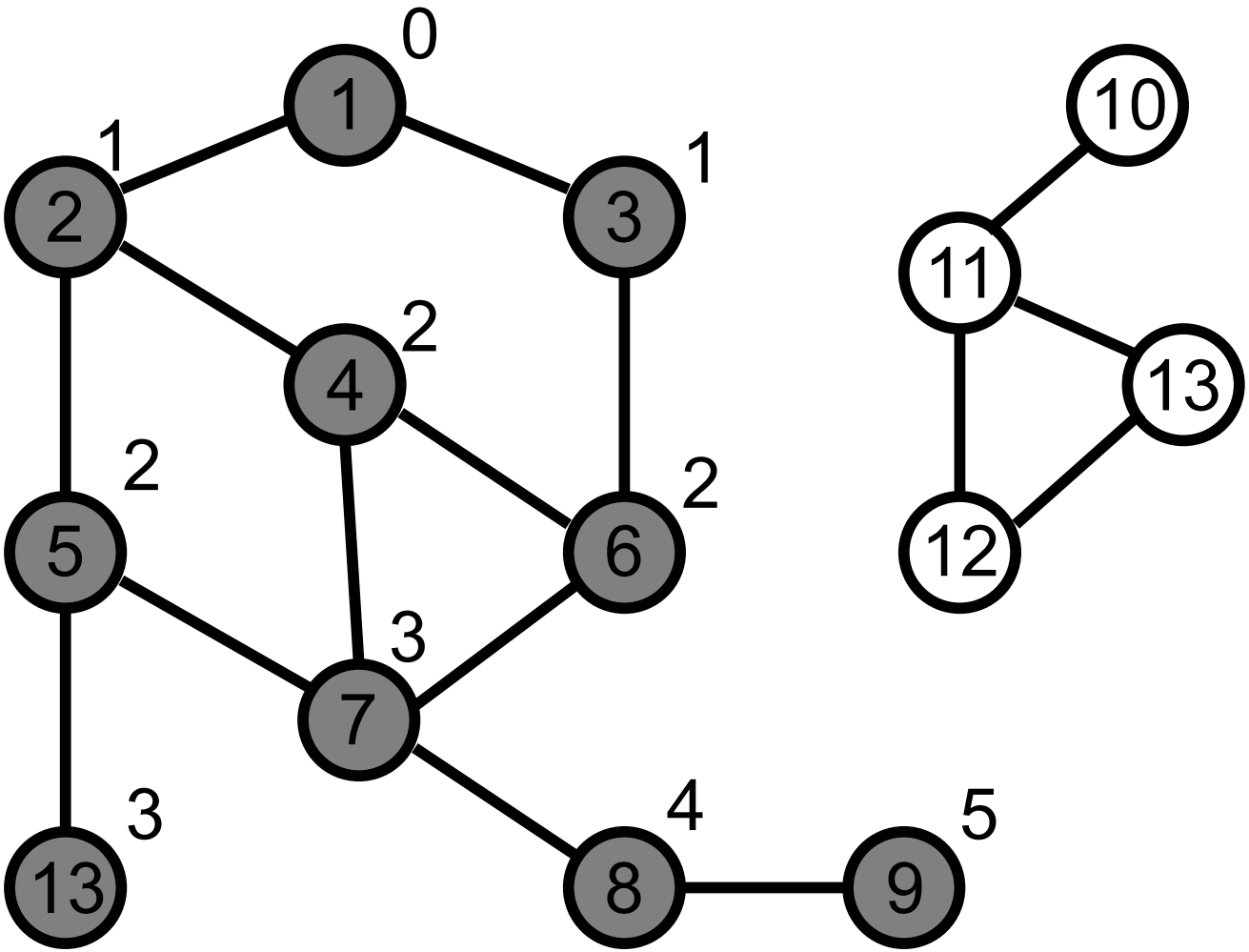
Przeglądanie wszerz



Przeglądanie wszerz



Przeglądanie wszerz



Podział na spójne składowe

- dla każdego wierzchołka chcemy określić, do której spójnej składowej należy (i przy okazji policzyć wszystkie spójne składowe)
- weź pierwszy wierzchołek i znajdź wszystkie wierzchołki, które są w tej samej składowej co on (wszerz lub w głąb)
- przypisz wszystkie znalezione wierzchołki do pierwszej składowej

Podział na spójne składowe

- weź pierwszy wierzchołek nie przypisany do żadnej składowej, i znajdź wszystkie wierzchołki, które są w tej samej składowej co on (wszerz lub w głąb)
- przypisz wszystkie znalezione wierzchołki do drugiej składowej
- powtarzaj aż nie uda się znaleźć nieprzypisanego wierzchołka
- tablicę *odw* można zastąpić numerami składowych:
  0 — wierzchołek nieodwiedzony
  i — wierzchołek należy do i-tej składowej

Algorytm Dijkstry

- najkrótsza ścieżka pomiędzy dwoma wierzchołkami
  lub od wybranego do wszystkich pozostałych
- etykiety krawędzi oznaczają odległości pomiędzy wierzchołkami
  (“długości” krawędzi)
- każdy wierzchołek posiada etykietę, która mówi o najkrótszej
  dotychczas znalezionej ścieżce do tego wierzchołka
- każdy wierzchołek może być ponadto zamknięty lub otwarty
- początkowo wszystkie wierzchołki są otwarte

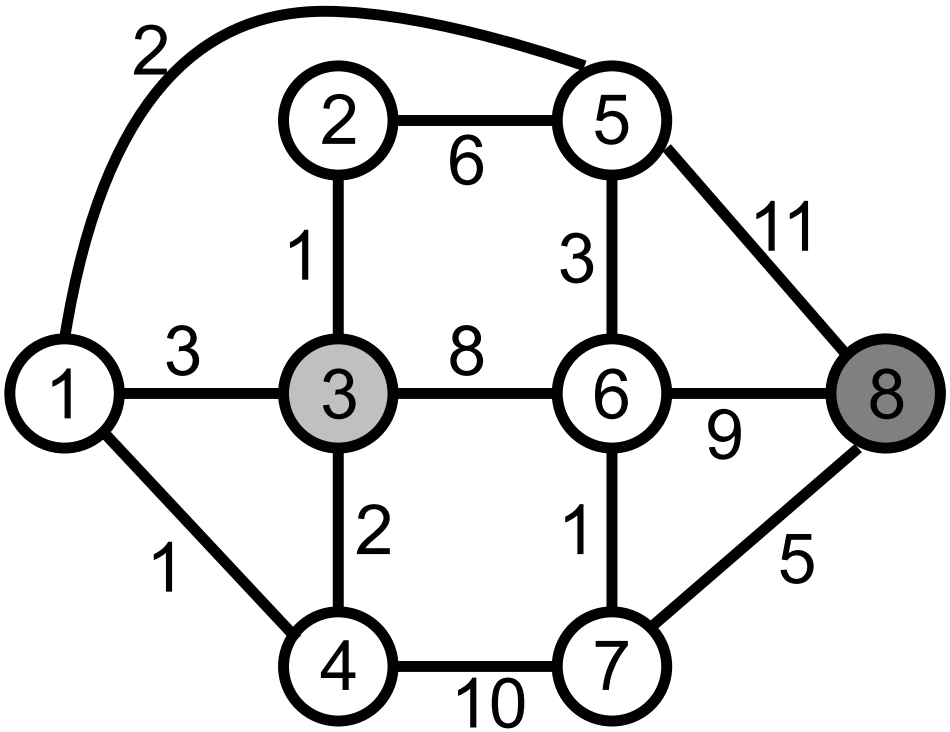
Algorytm Dijkstry

- w każdym kroku wybieramy otwarty wierzchołek z najmniejszą etykietą, zamykamy go i przeglądamy jego sąsiadów, sprawdzając, czy można do nich dojść krótszą drogą
- przerywamy, gdy zamkniemy wierzchołek docelowy (lub gdy zamkniemy wszystkie wierzchołki, jeżeli interesują nas odległości do wszystkich wierzchołków)
- zamknięcie wierzchołka = znamy długość najkrótszej drogi do tego wierzchołka

Algorytm Dijkstry

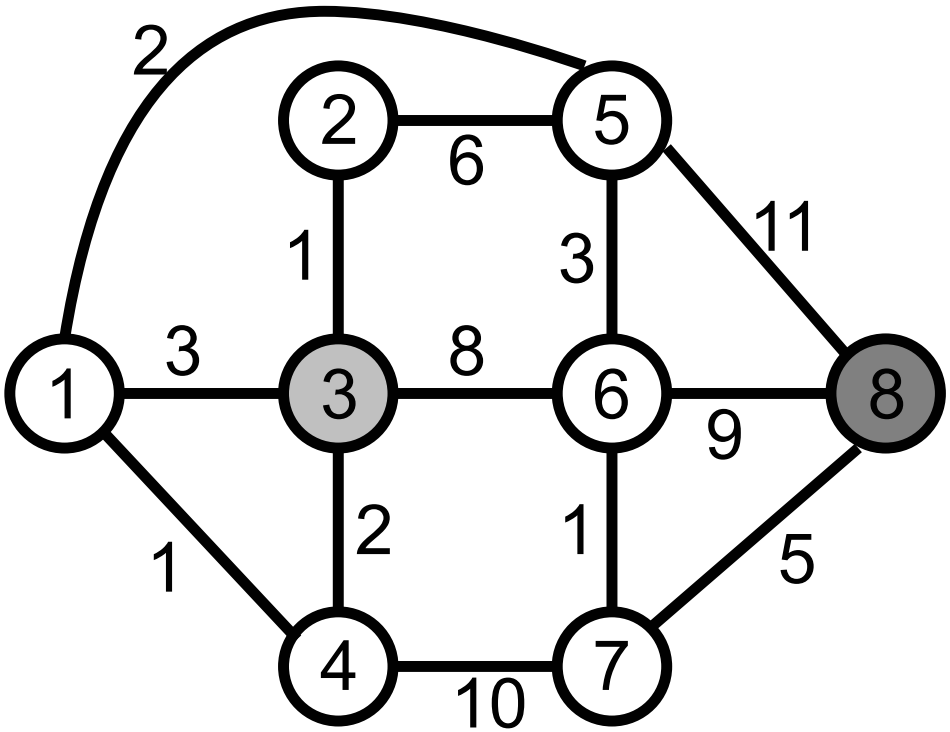
for $i = 1, \ldots, n$ **do** $E[i] = +\infty$
for $i = 1, \ldots, n$ **do** $Zamkniety[i] = False$
$E[v_s] = 0$
while $Zamkniety[v_d] = False$ **do**
    v – niezamknięty wierzchołek o najmniejszej etykiecie
    $Zamkniety[v] = True$
    for $u =$ każdy sąsiad v **do**
        if $Zamkniety[u]$ **then continue**
        if $E[v] + D(v, u) < E[u]$ **then**
            $E[u] = E[v] + D(u, v)$
        end if
    end for
end while

Algorytm Dijkstry



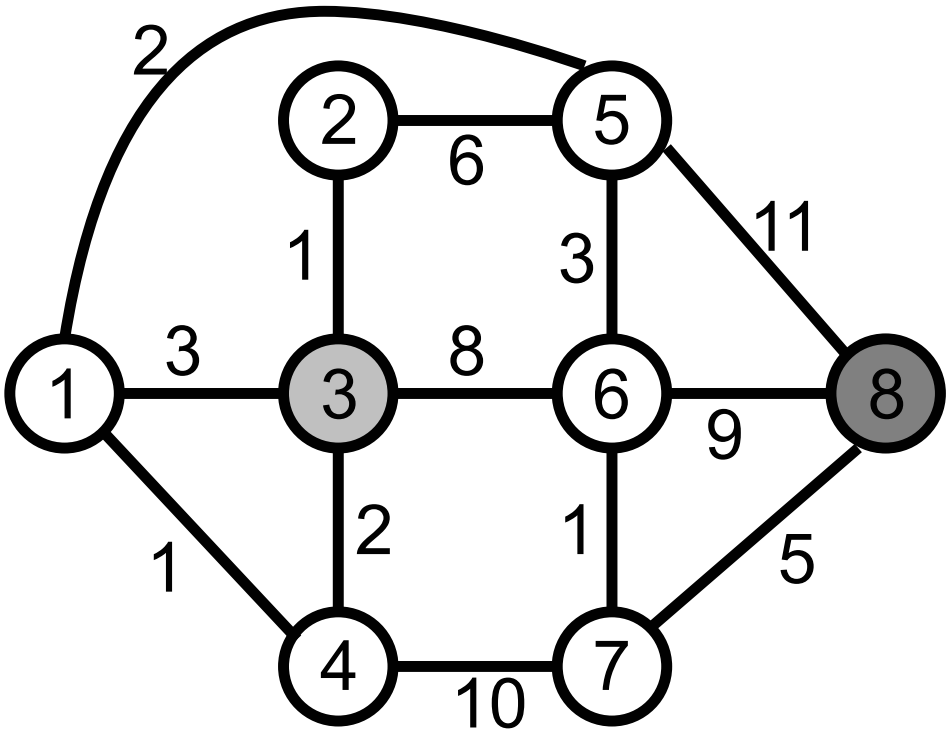
1	2	3	4	5	6	7	8
∞	∞	∞	∞	∞	∞	∞	∞

Algorytm Dijkstry



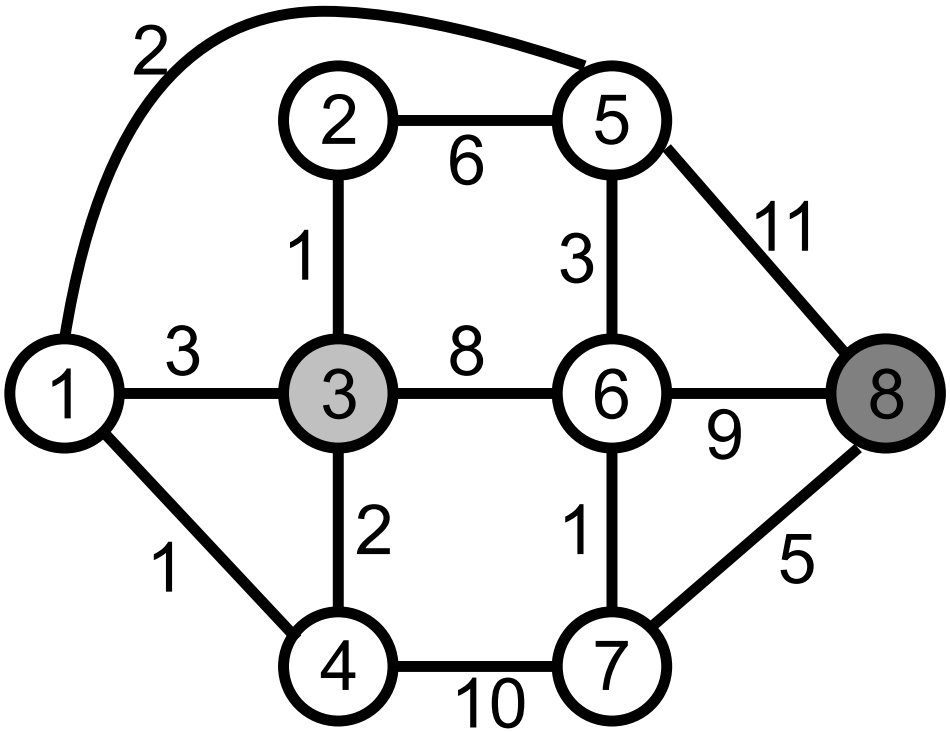
1	2	3	4	5	6	7	8
∞	∞	0	∞	∞	∞	∞	∞

Algorytm Dijkstry



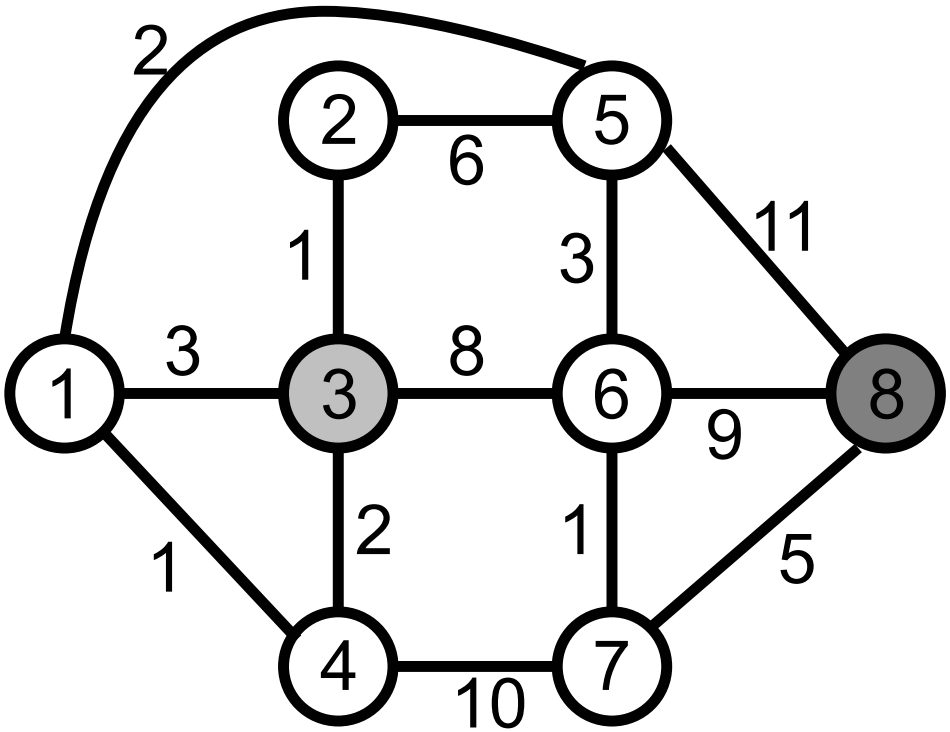
1	2	3	4	5	6	7	8
3	1	0	2	∞	8	∞	∞

Algorytm Dijkstry



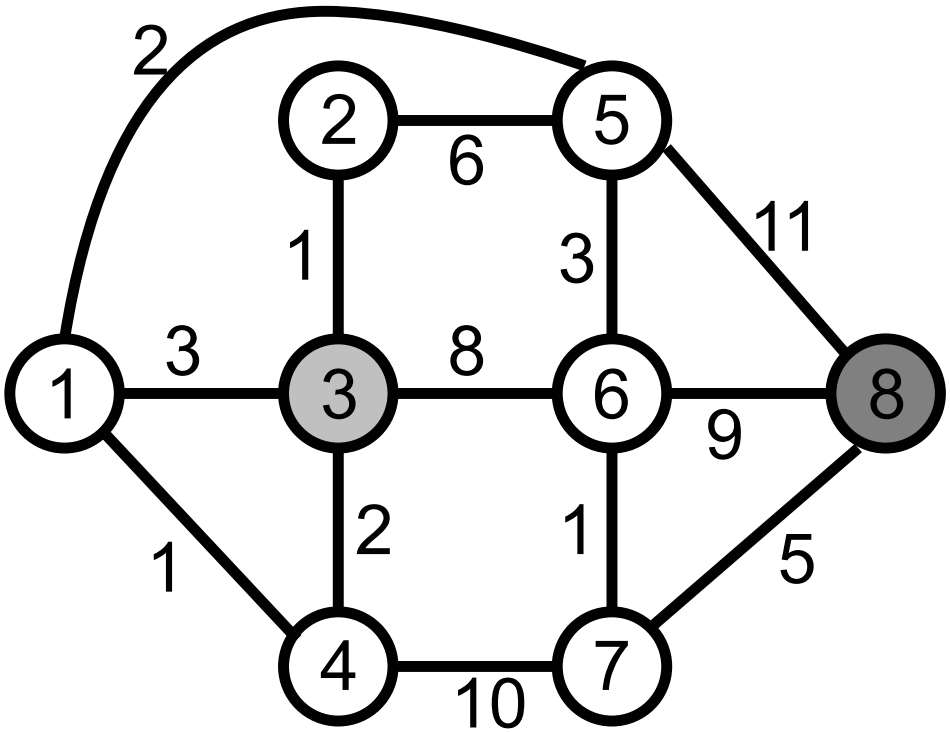
1	2	3	4	5	6	7	8
3	1	0	2	7	8	∞	∞

Algorytm Dijkstry



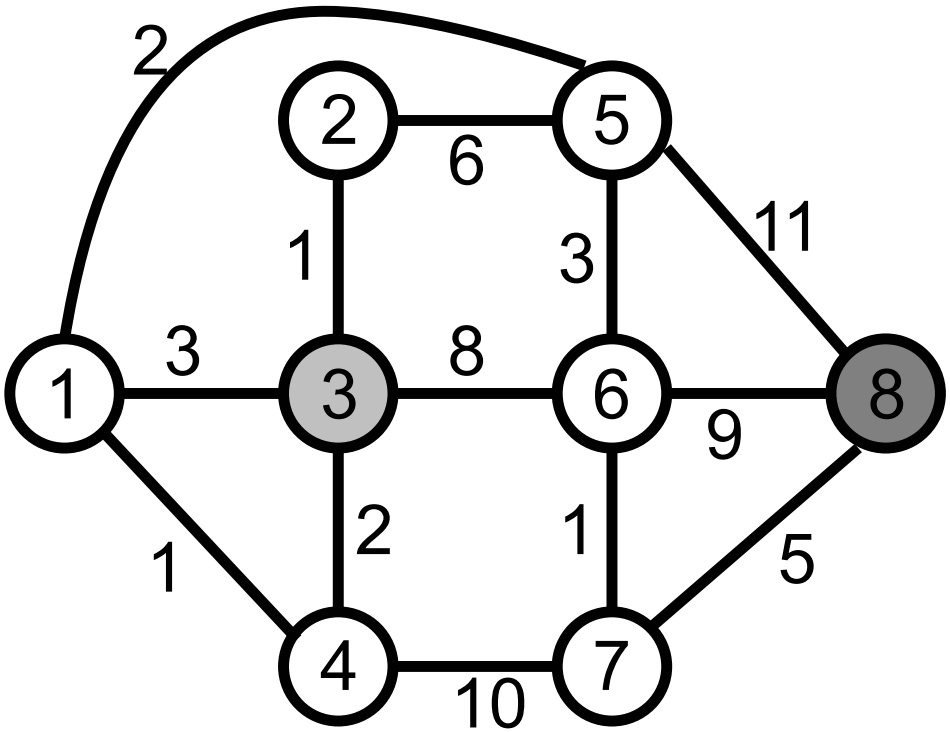
1	2	3	4	5	6	7	8
3	1	0	2	7	8	12	∞

Algorytm Dijkstry



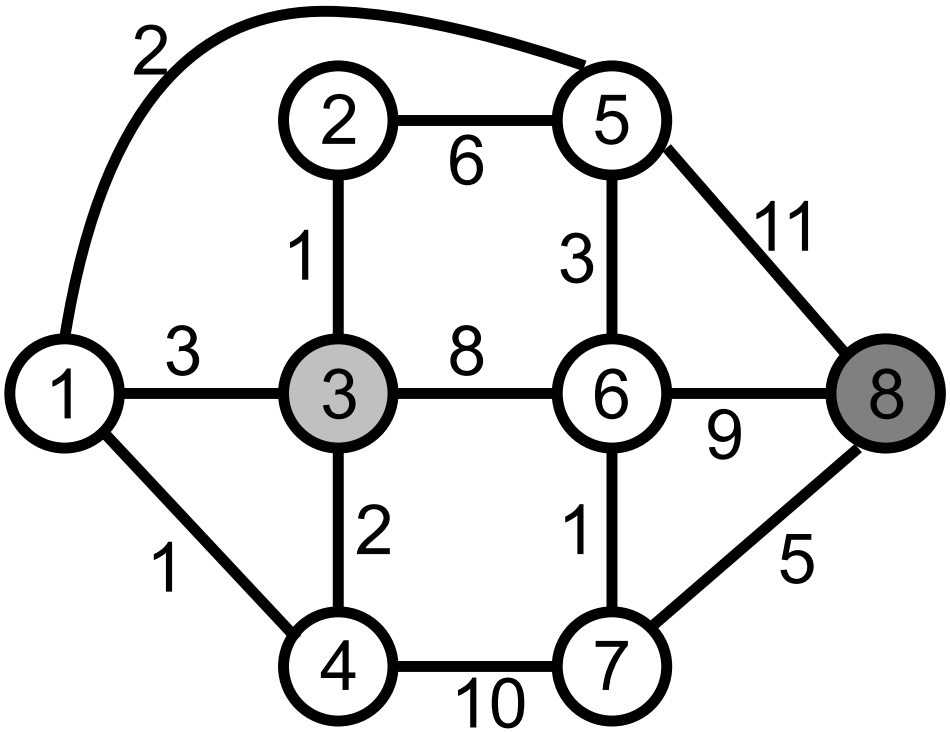
1	2	3	4	5	6	7	8
3	1	0	2	5	8	12	∞

Algorytm Dijkstry



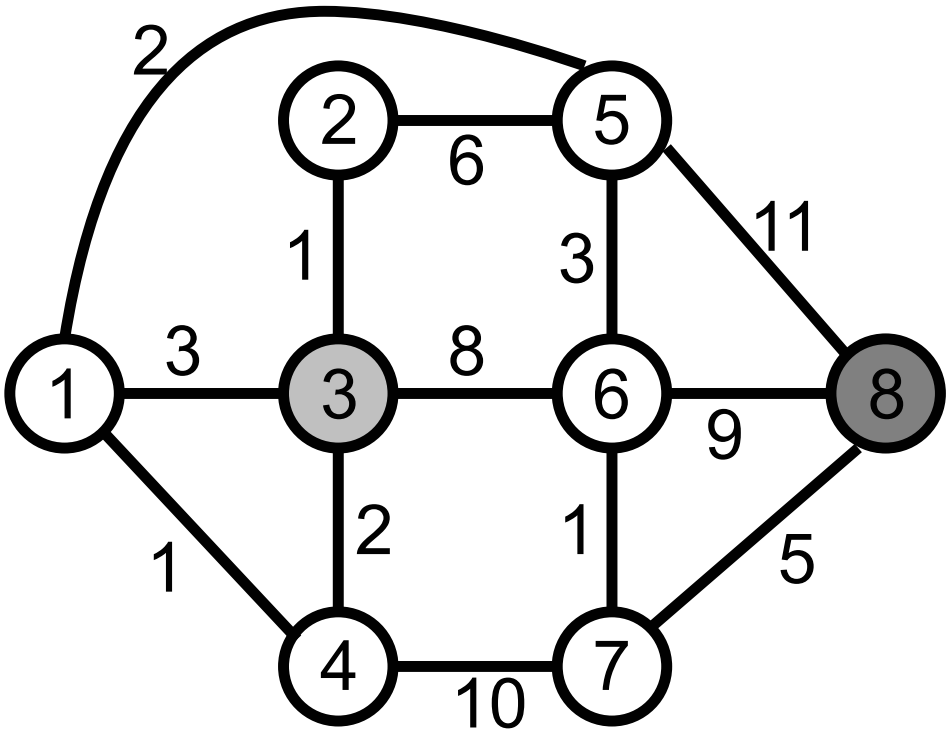
1	2	3	4	5	6	7	8
3	1	0	2	5	8	12	16

Algorytm Dijkstry



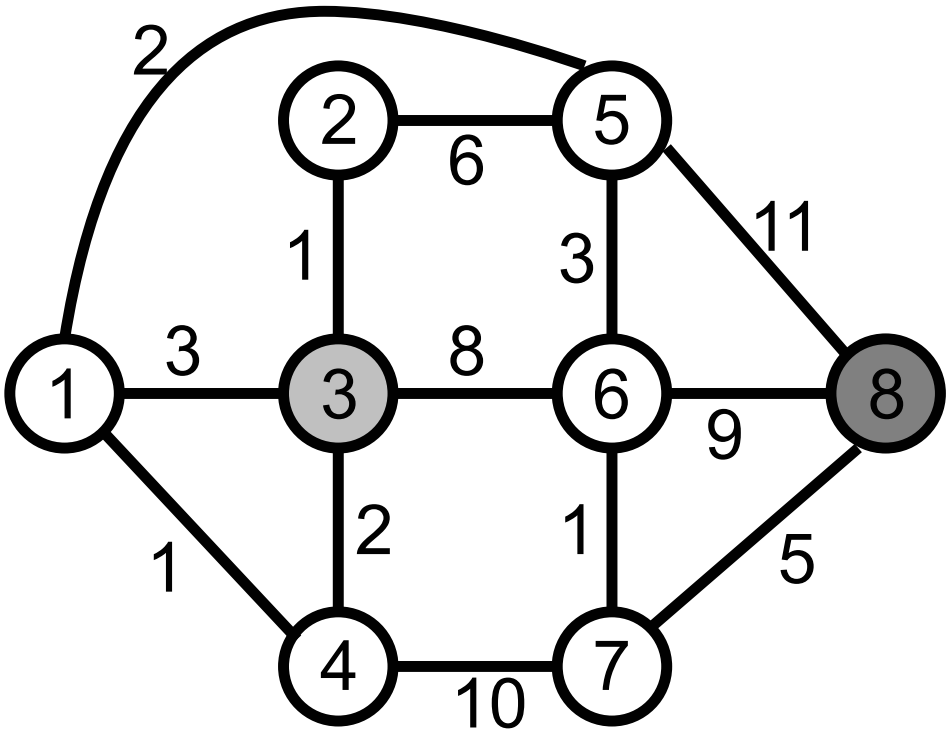
1	2	3	4	5	6	7	8
3	1	0	2	5	8	9	16

Algorytm Dijkstry



1	2	3	4	5	6	7	8
3	1	0	2	5	8	9	14

Algorytm Dijkstry



1	2	3	4	5	6	7	8
3	1	0	2	5	8	9	14

Algorytm Dijkstry

- po zamknięciu, etykieta wierzchołka opisuje długość najkrótszej ścieżki z wierzchołka startowego do danego
- jeżeli interesuje nas przebieg ścieżki, możemy dodatkowo pamiętać, dla każdego wierzchołka, z którego wierzchołka do niego doszliśmy (robimy to w momencie aktualizacji etykiety)
- możemy także odtworzyć ścieżkę na podstawie etykiet:
  - ostatni na ścieżce będzie wierzchołek docelowy
  - przedostatni będzie ten, którego odległość od wierzchołka początkowego, powiększona o długość krawędzi do ostatniego jest równa długości całej ścieżki
  - ...

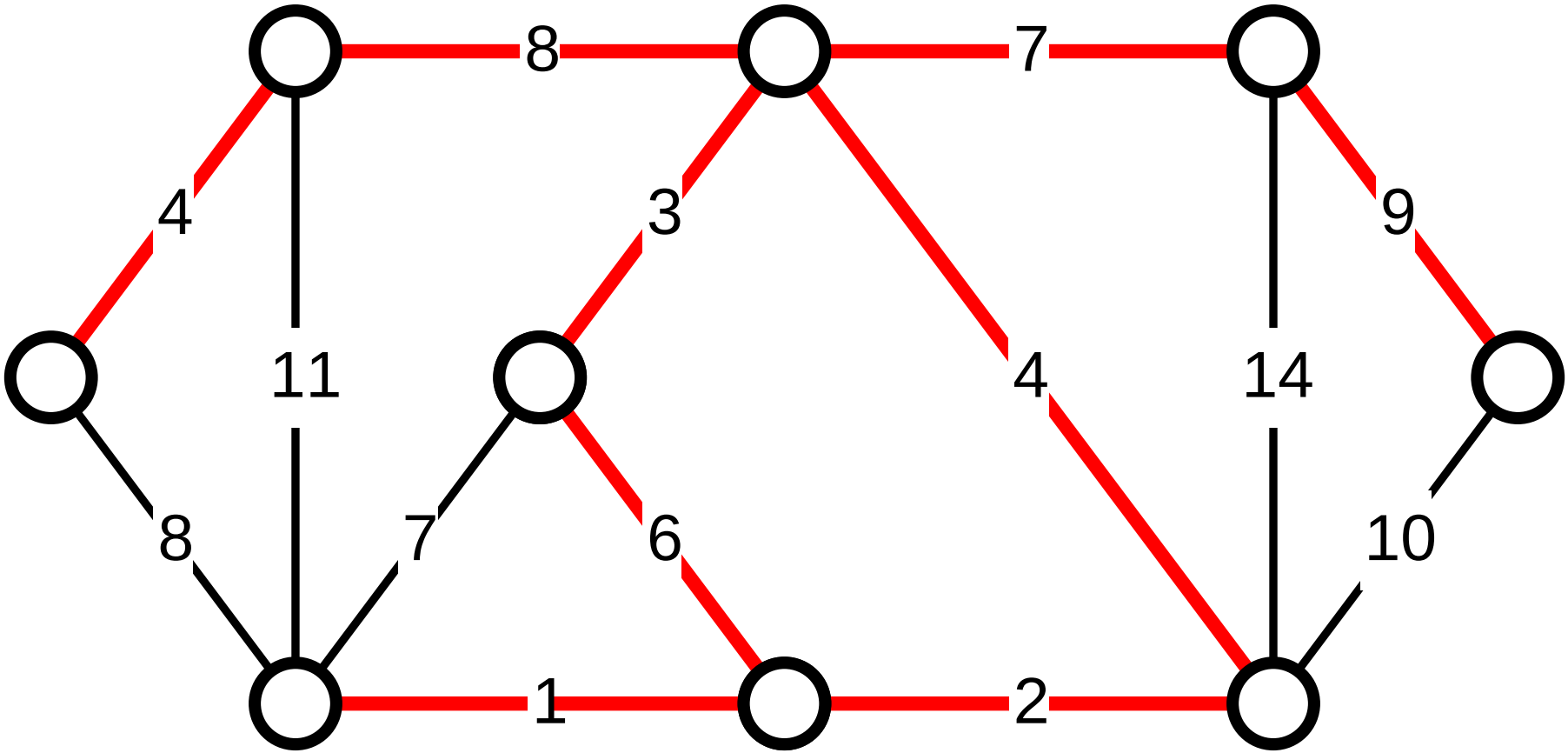
Algorytm Dijkstry — złożoność

- $O(n^2 + m)$
- $O(n \log n + m \log n)$
- ...

Minimalne drzewo spinające

- w danym grafie G, z wagami na krawędziach, chcemy znaleźć zbiór krawędzi F spełniający:
  - każdy wierzchołek sąsiaduje z jakąś krawędzią w F
  - krawędzie z F tworzą drzewo
  - suma wag krawędzi z F jest jak najmniejsza
- np. chcemy połączyć wszystkie miasta siecią elektryczną o najmniejszym koszcie (położyć jak najmniejszą łączną długość przewodów)

Minimalne drzewo spinające



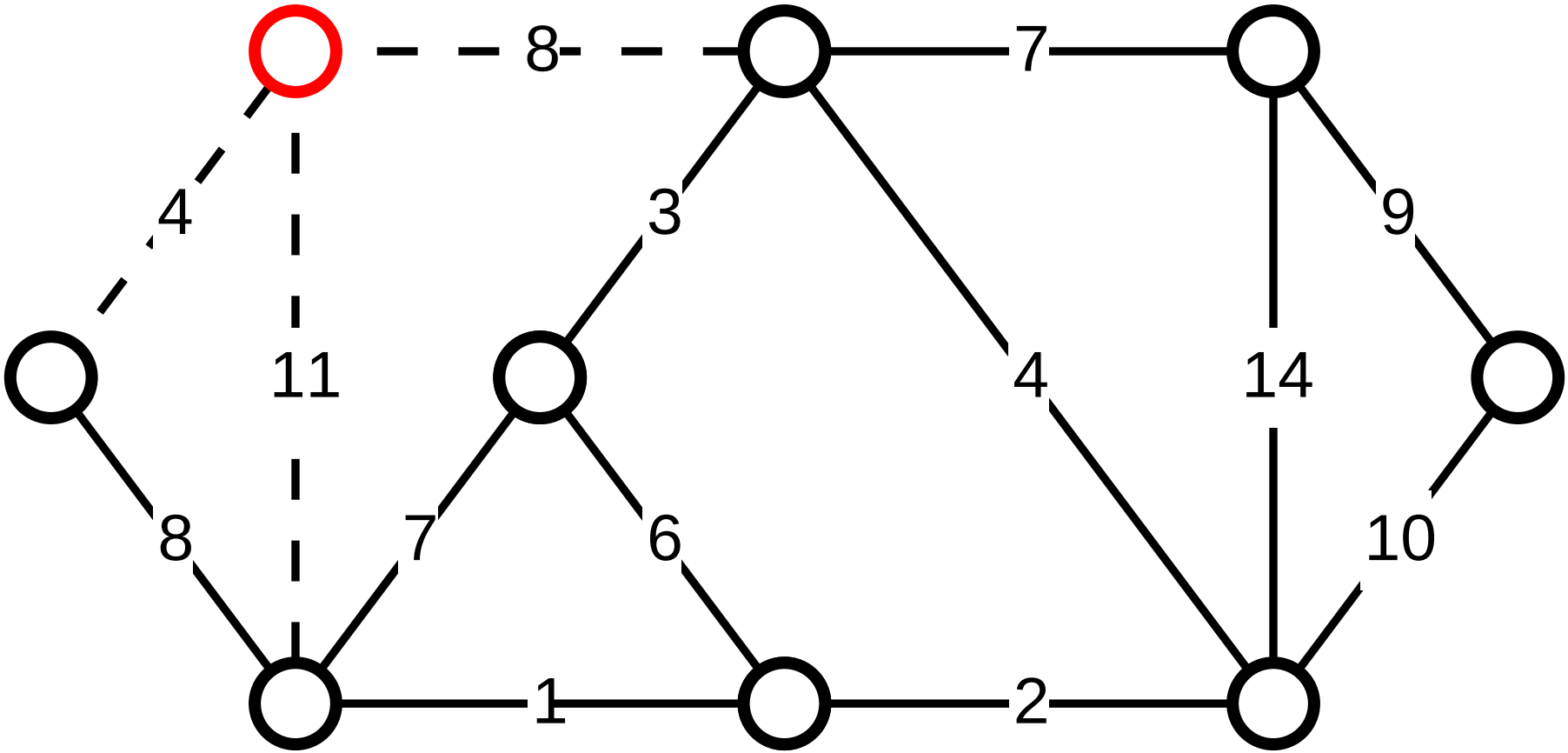
Algorytm Prima

- rozpoczynamy od dowolnego wierzchołka
- w każdym kroku wybieramy krawędź o najmniejszej wadze, która sąsiaduje z jednej strony z dotychczasowym drzewem, z drugiej strony z wierzchołkiem spoza drzewa
- powtarzamy operację $n - 1$ razy
- w przypadku grafów niespójnych (tworząc las spinający) musimy podzielić graf na spójne składowe i w każdej z nich osobno poszukać drzewa spinającego
- *kopiec!*

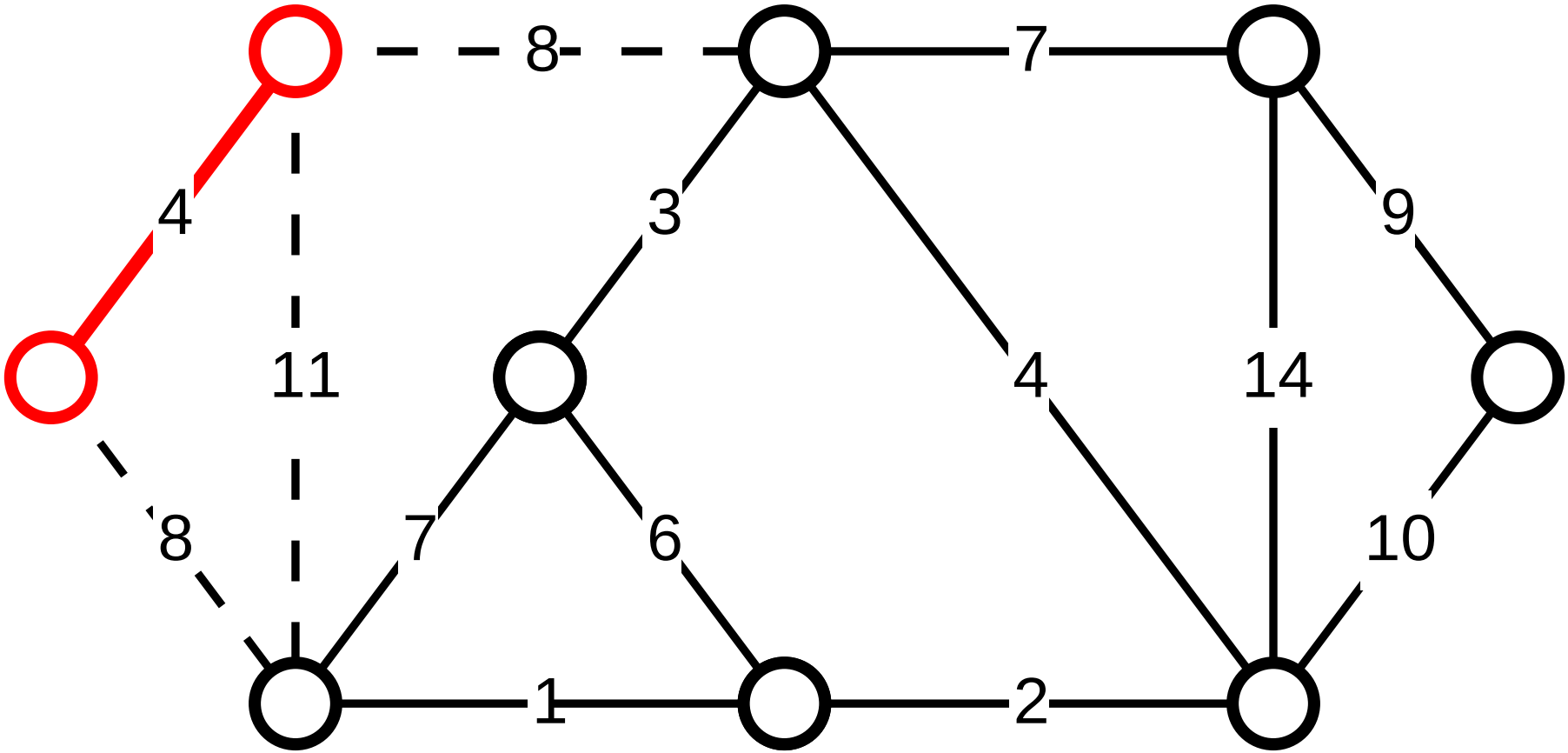
Algorytm Prima

$D = \{v_1\}$ — drzewo spinające
$K =$ pusty kopiec
dodaj do K $N(v_1)$ t.j. krawędzie sąsiadujące z D
for $i = 1, \ldots, n-1$ **do**
    $\{u, v\} =$ minimum z K i usuń ją z K
    if $u \in D$ i $v \in D$ **then continue**
    dodaj v i u, v do D
    dodaj $N(v)$ do K
end for

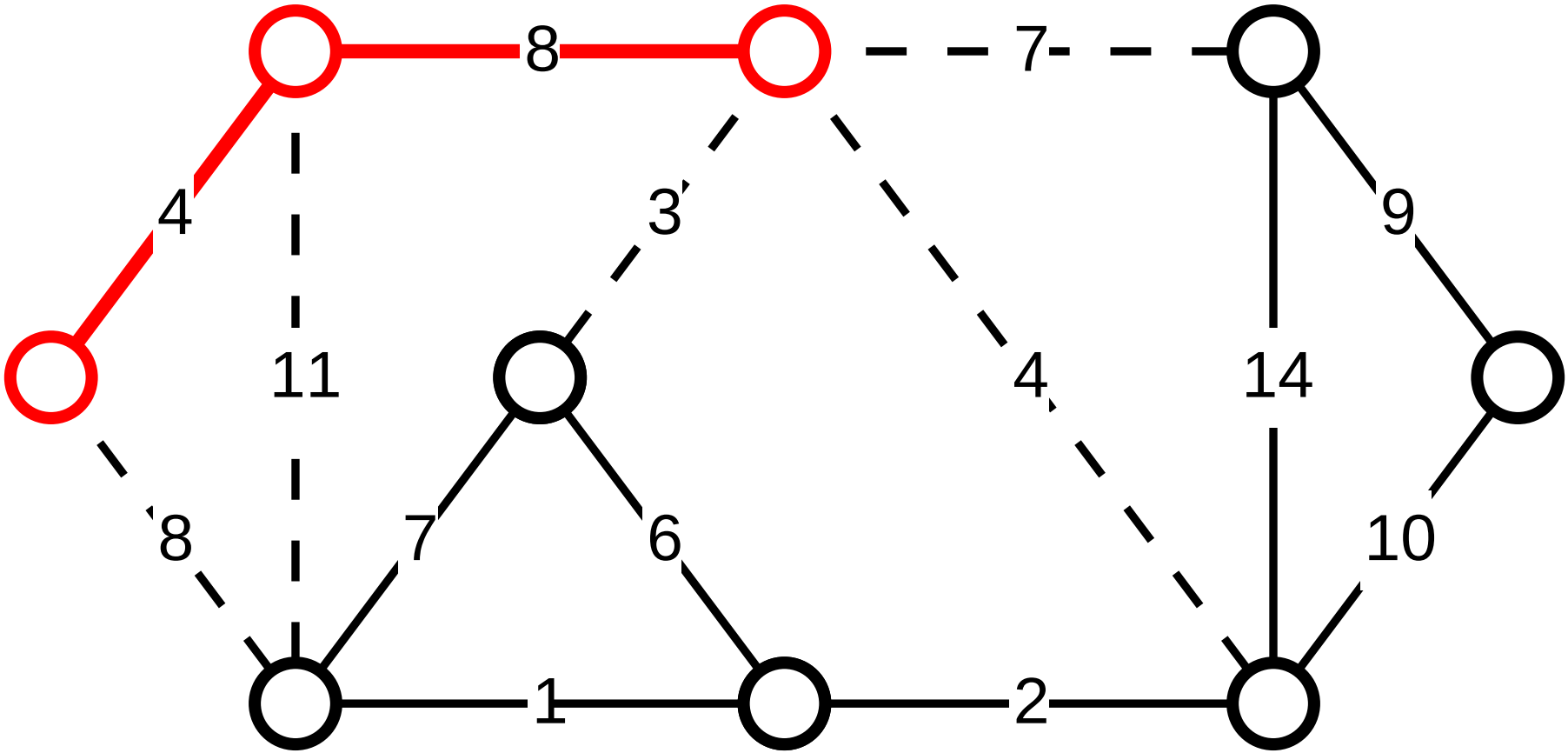
Algorytm Prima



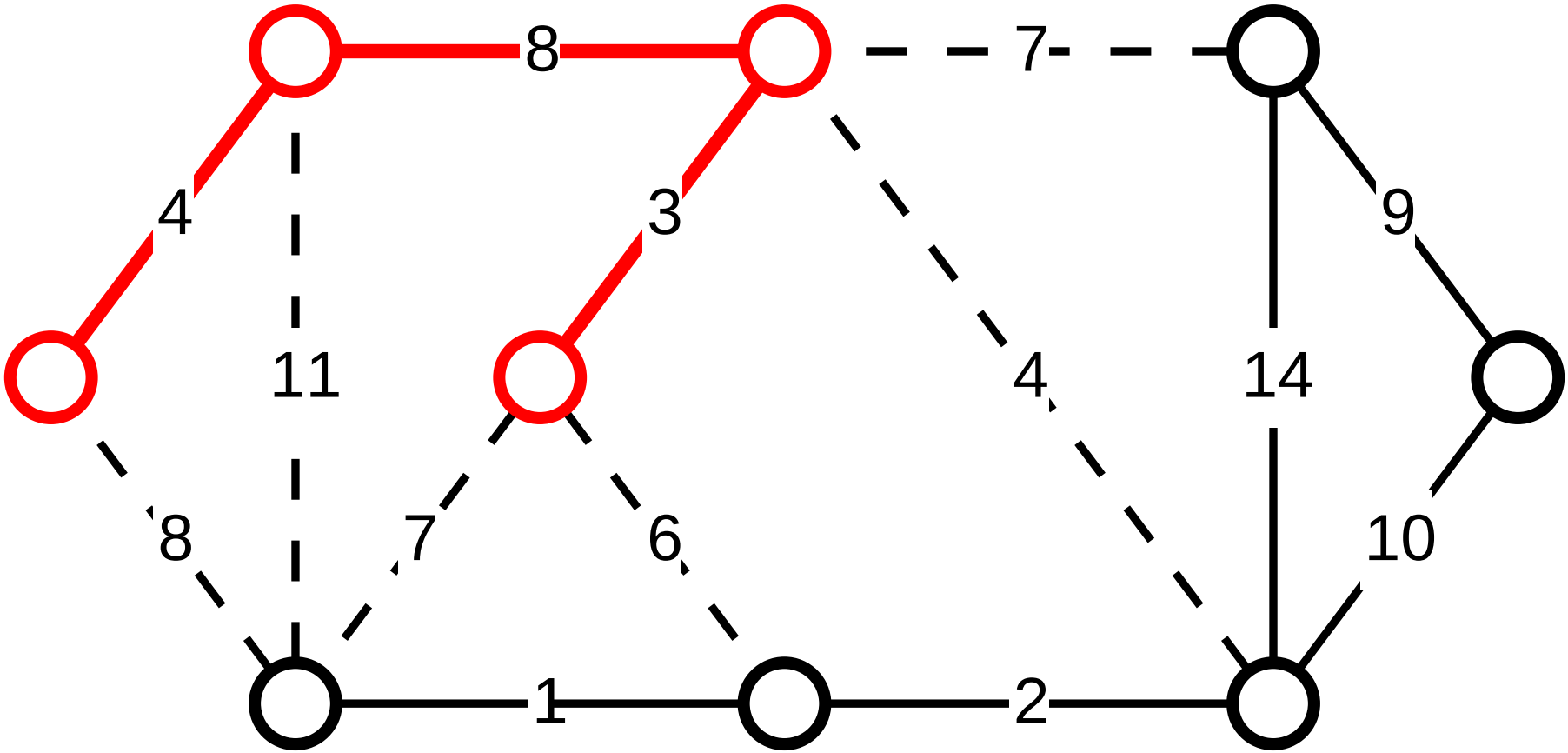
Algorytm Prima



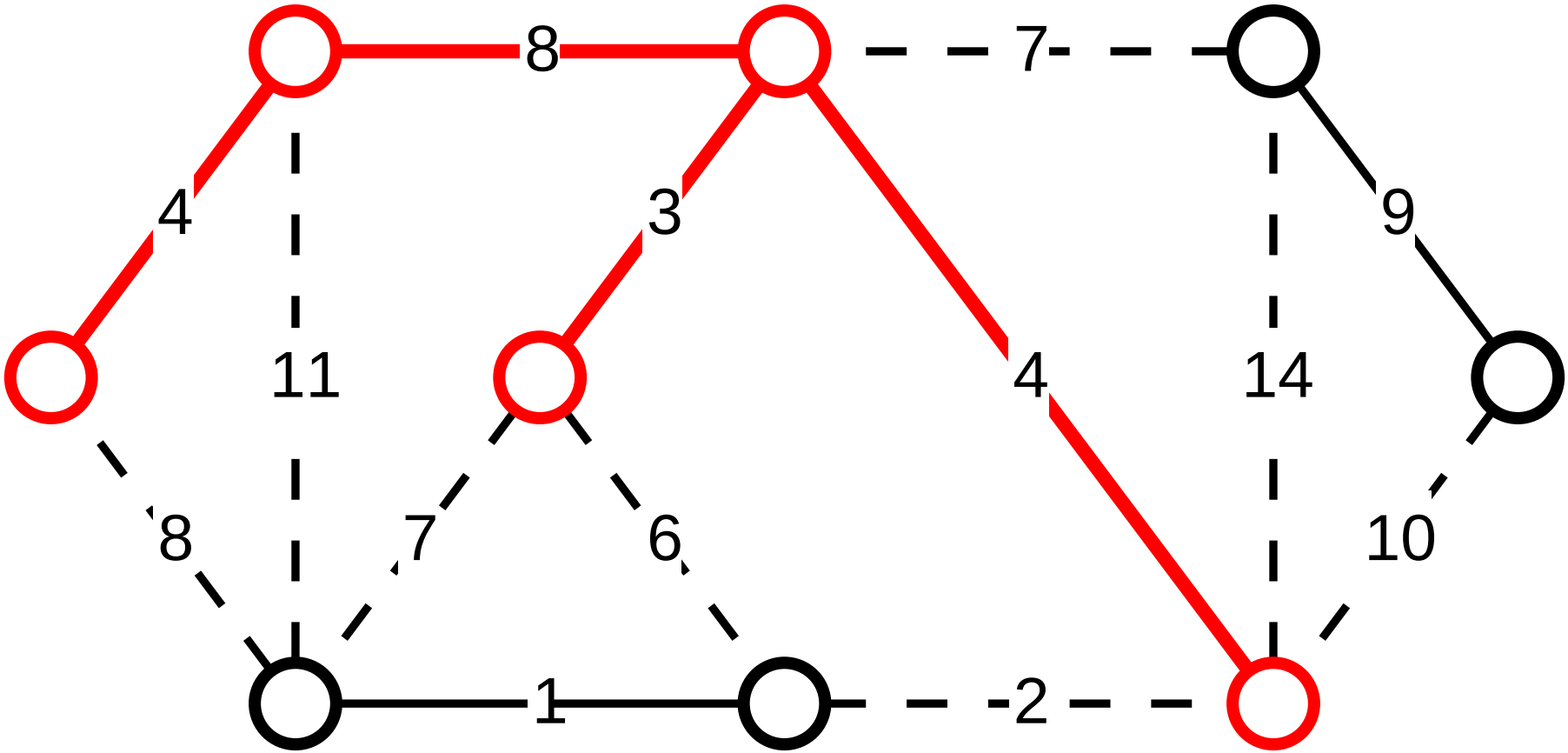
Algorytm Prima



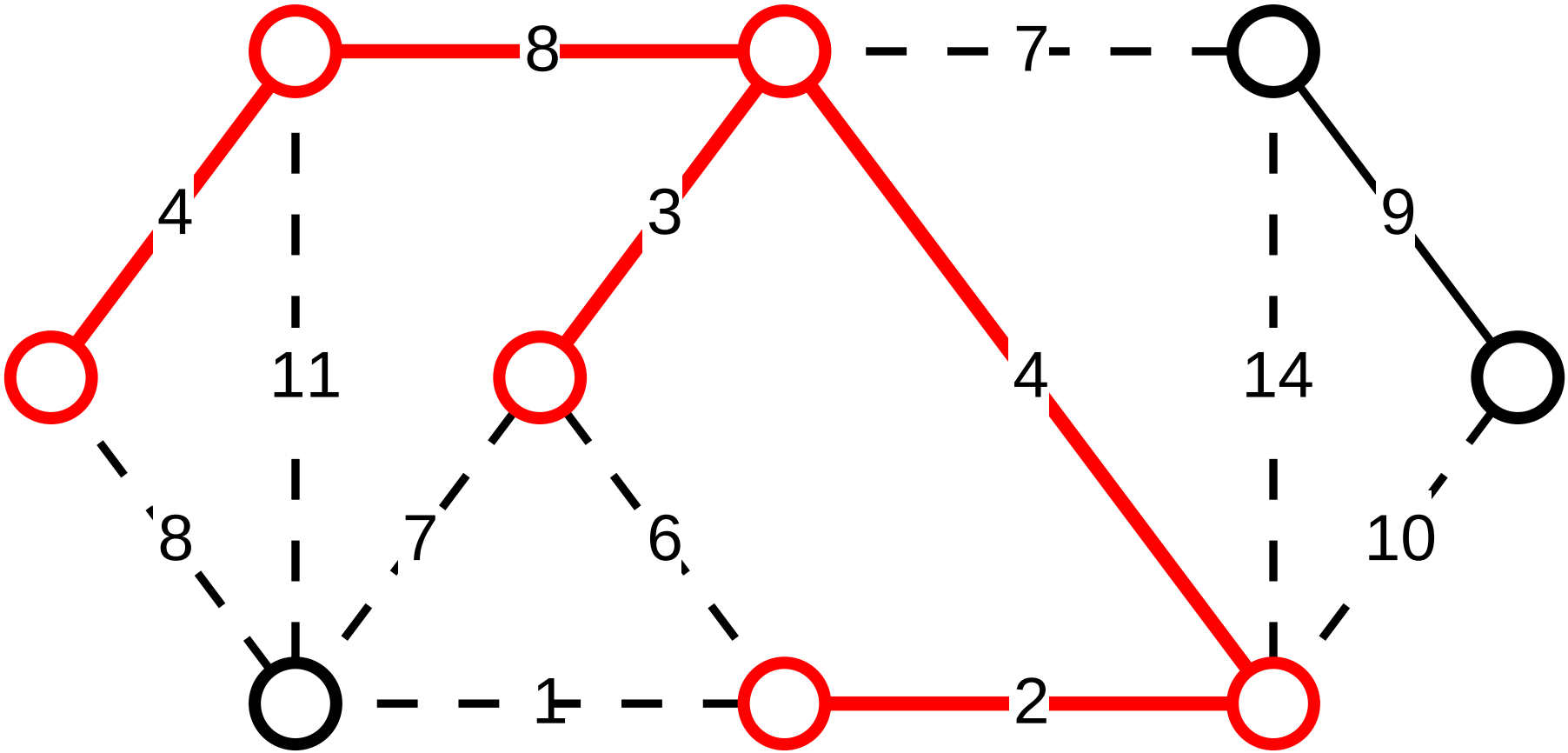
Algorytm Prima



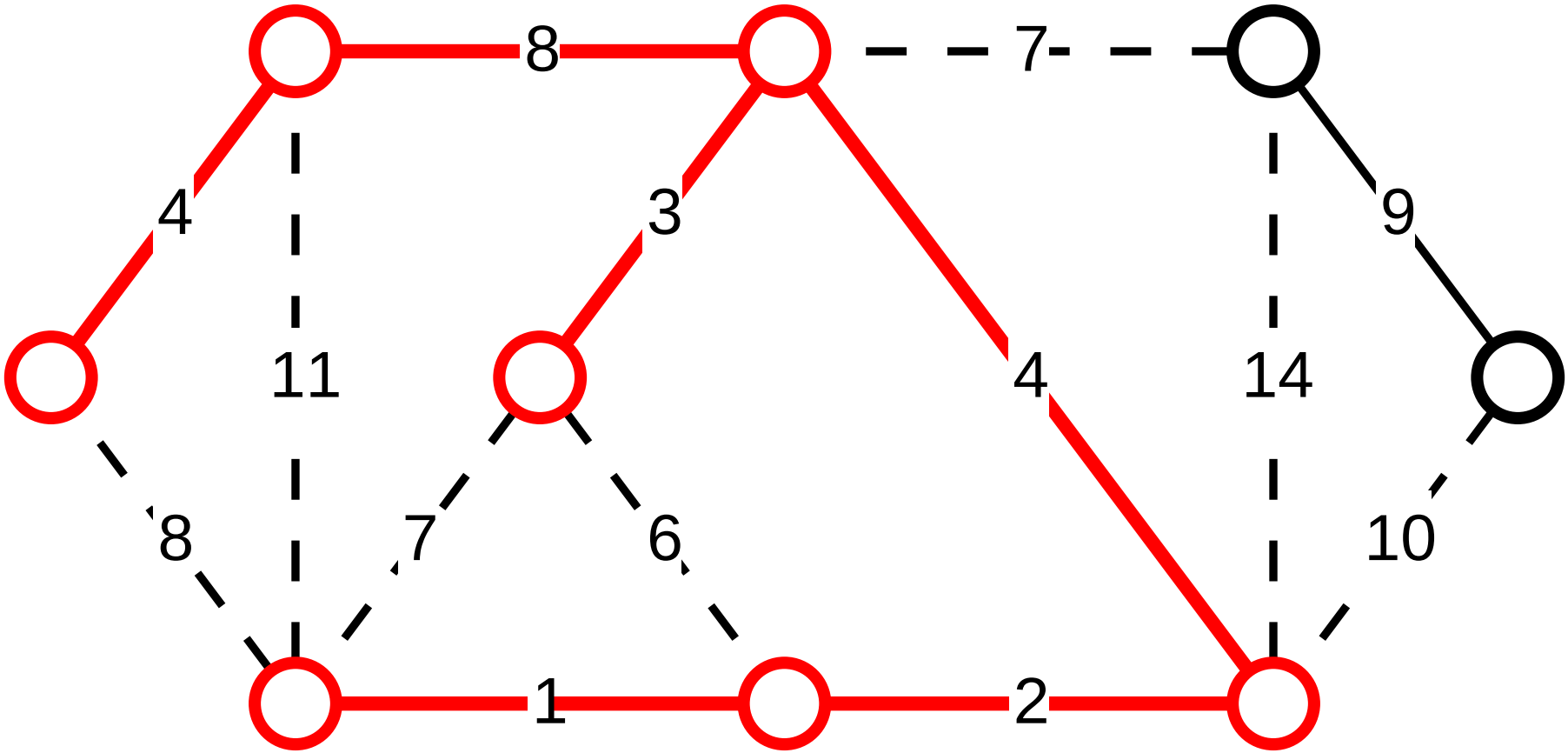
Algorytm Prima



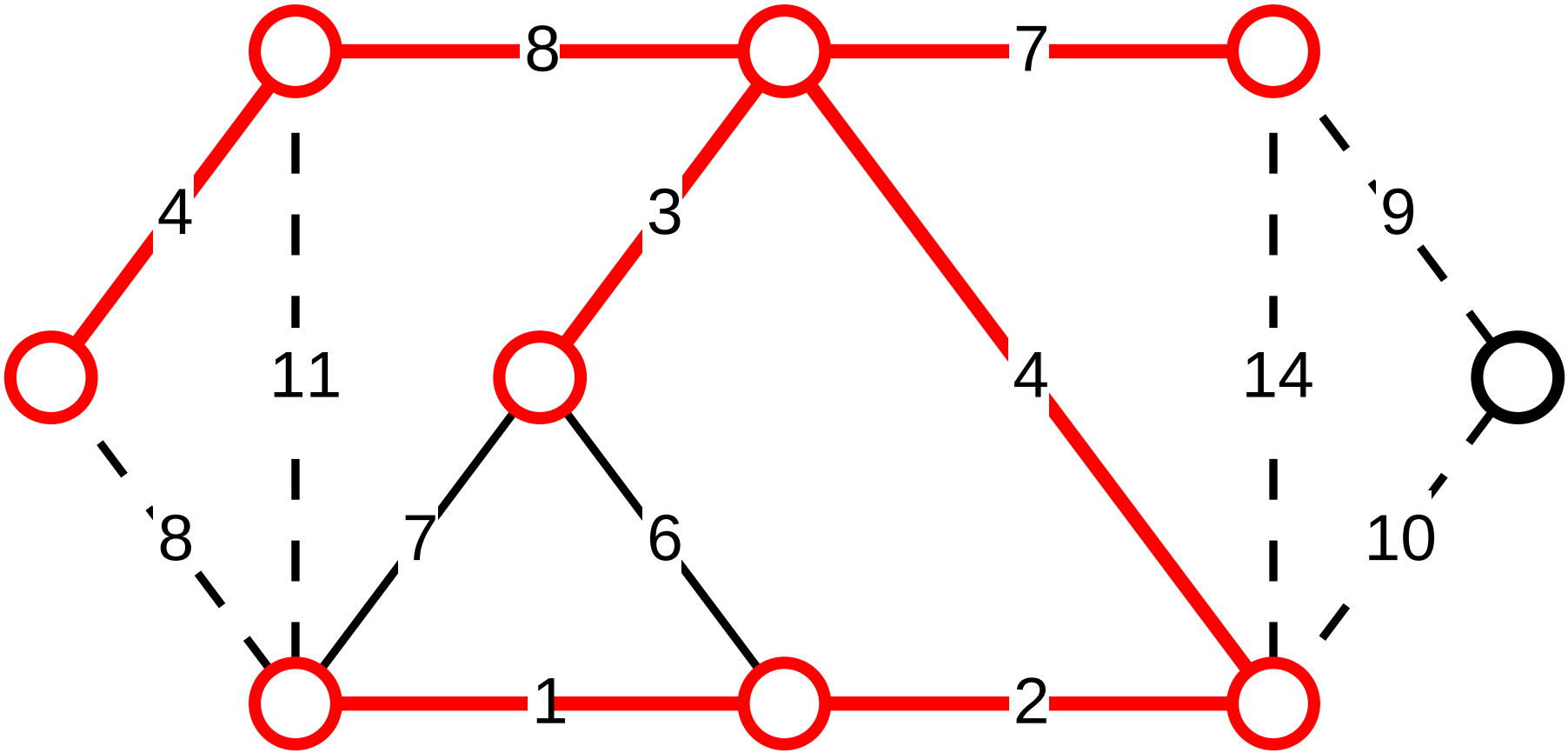
Algorytm Prima



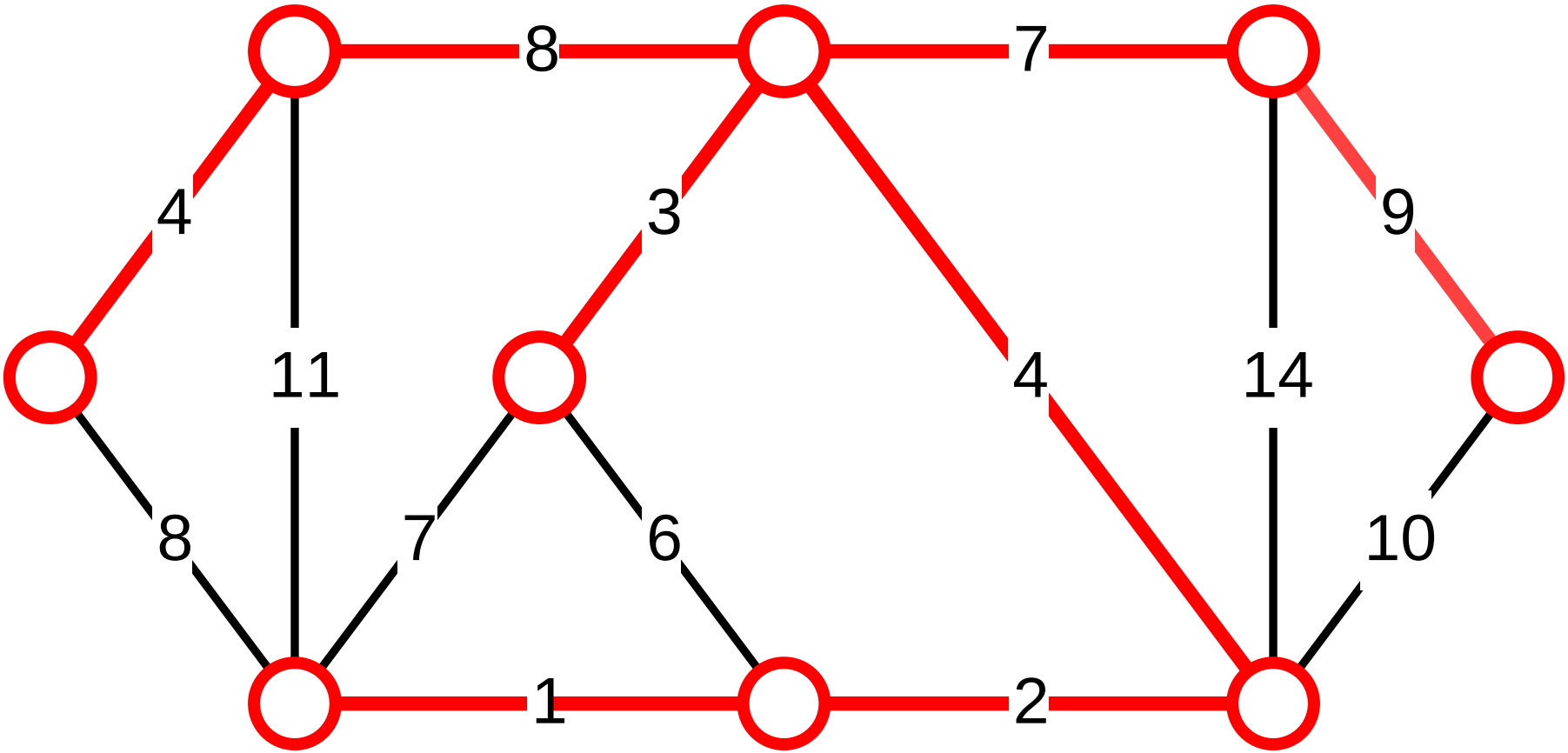
Algorytm Prima



Algorytm Prima



Algorytm Prima



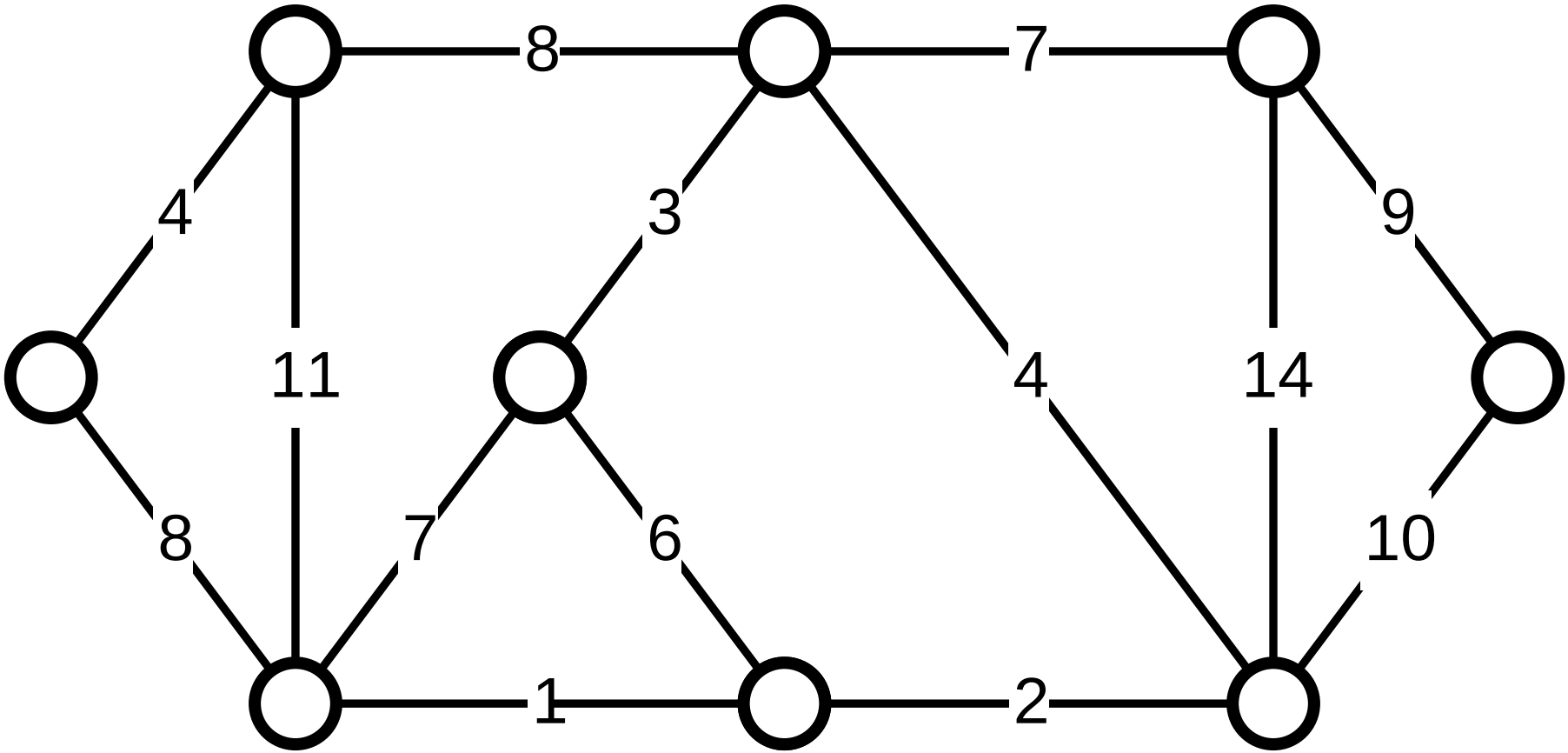
Algorytm Kruskala

- rozpoczynamy od pustego drzewa
- przeglądamy kolejno krawędzie zgodnie z niemalejącymi wagami
- jeżeli możemy dodać krawędź do drzewa (nie utworzy nam cyklu) — dodajemy ją
- nie wymaga dodatkowych operacji w przypadku grafów niespójnych

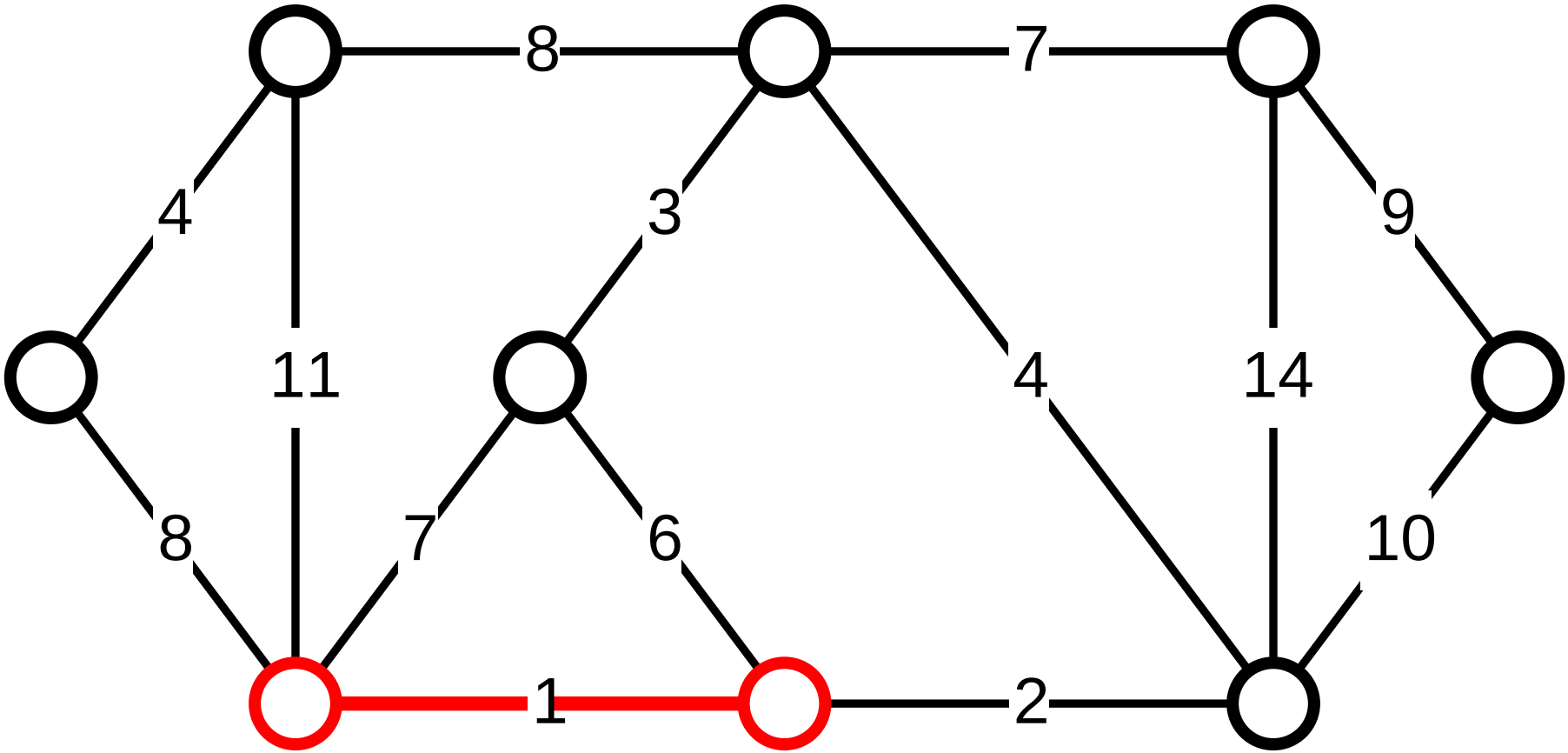
Algorytm Kruskala

$D =$ puste drzewo spinające
for $e =$ krawędzie w kolejności niemalejących wag **do**
    jeżeli dodanie e do D nie stworzy cyklu, dodaj e do D
    jeżeli D zawiera $n - 1$ krawędzi, przerwij
end for

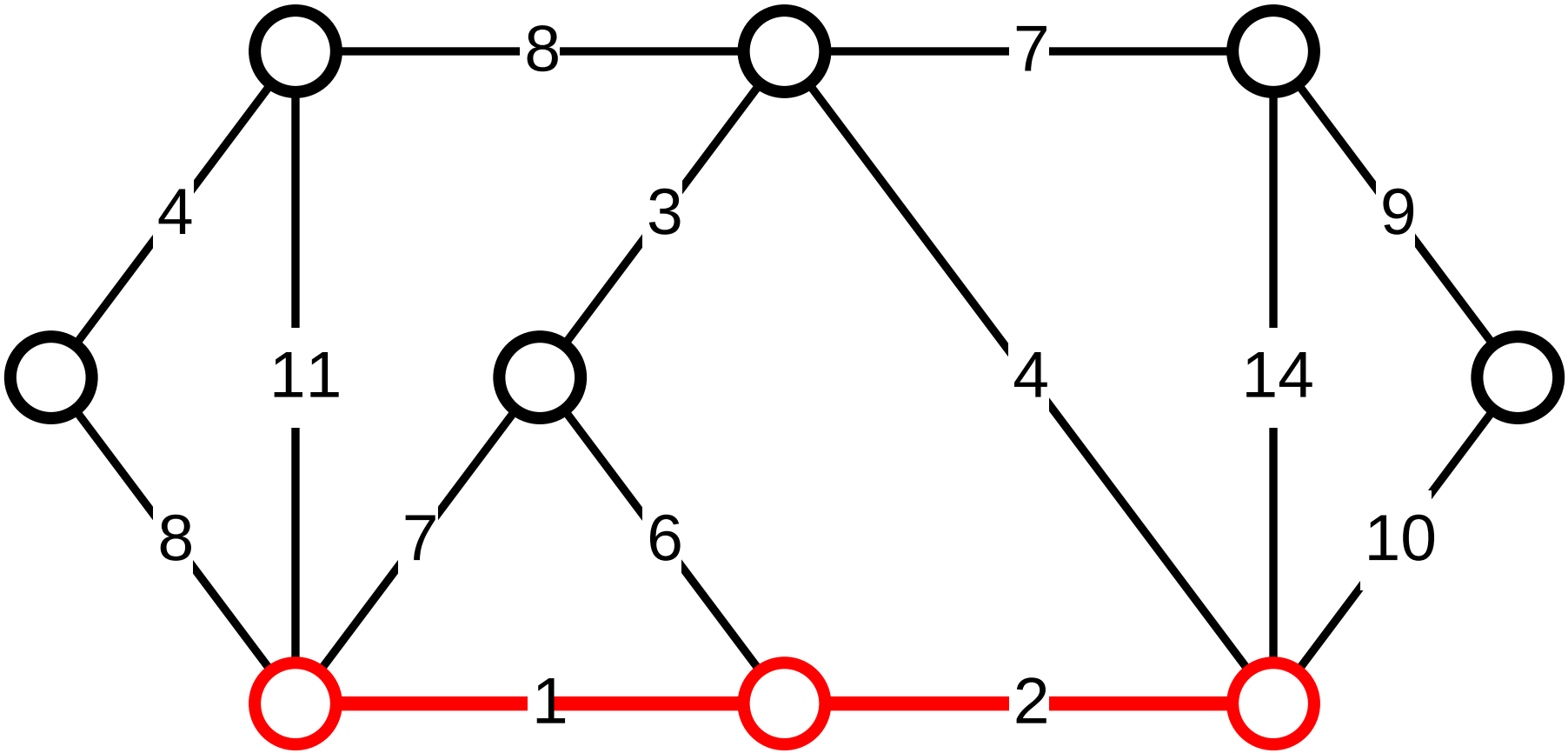
Algorytm Kruskala



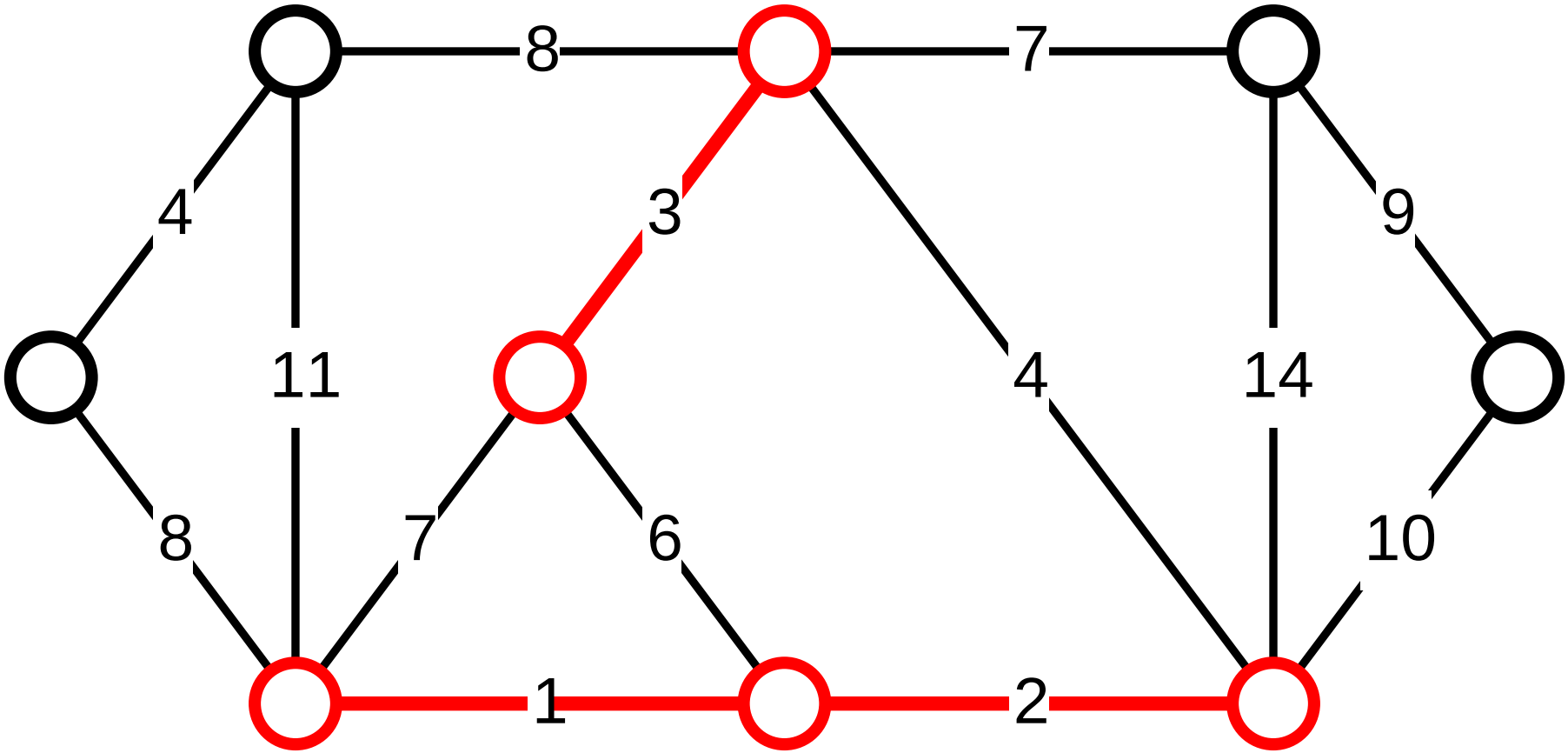
Algorytm Kruskala



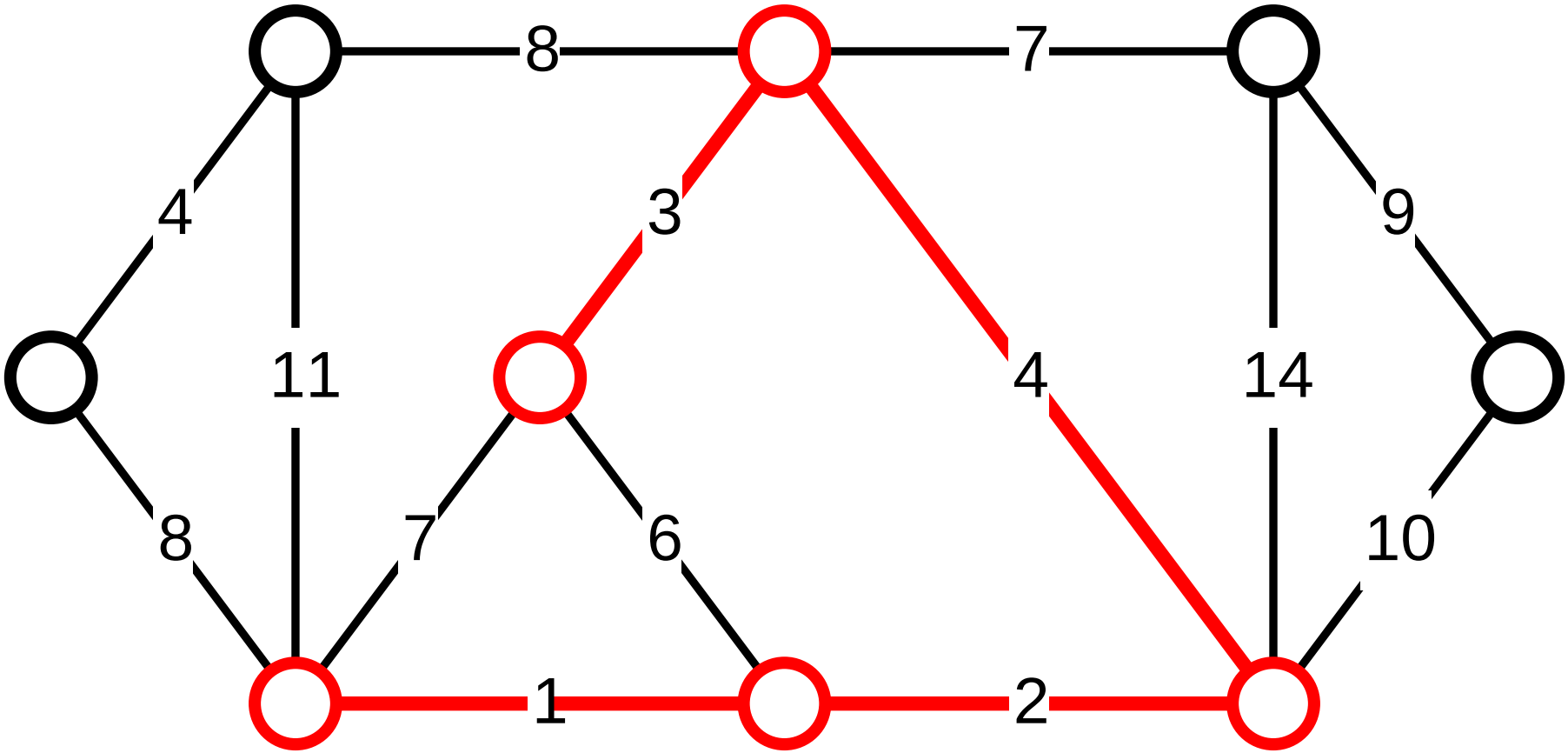
Algorytm Kruskala



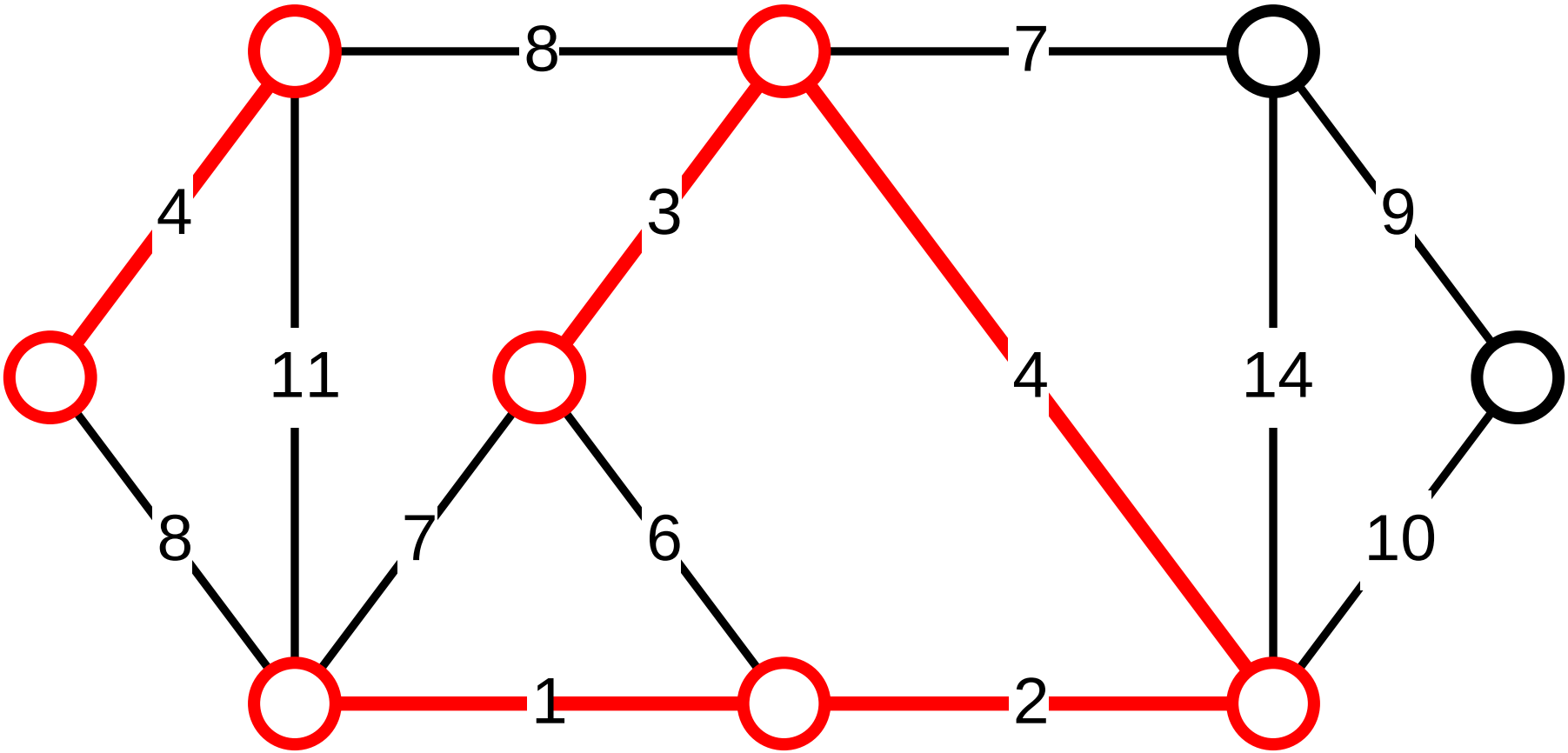
Algorytm Kruskala



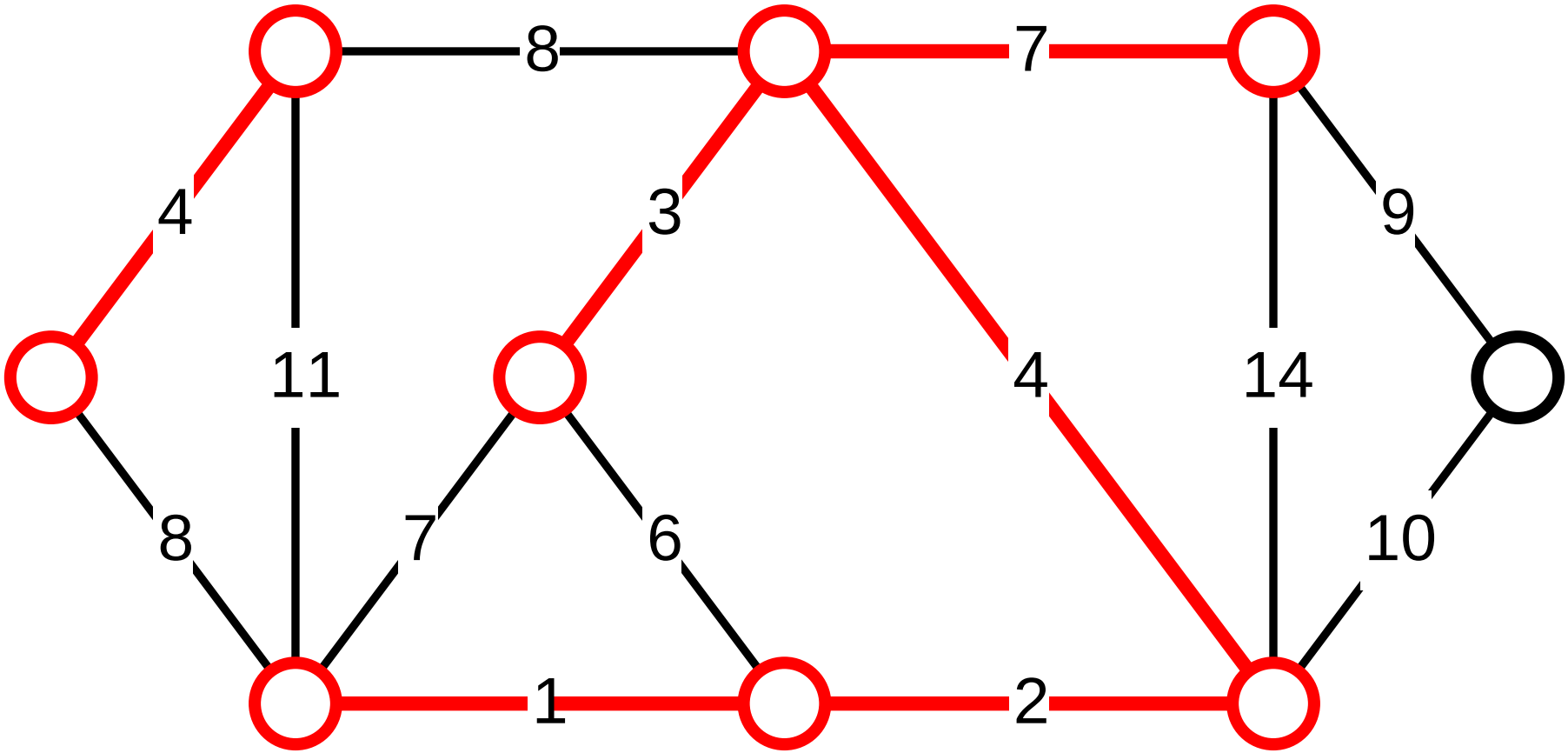
Algorytm Kruskala



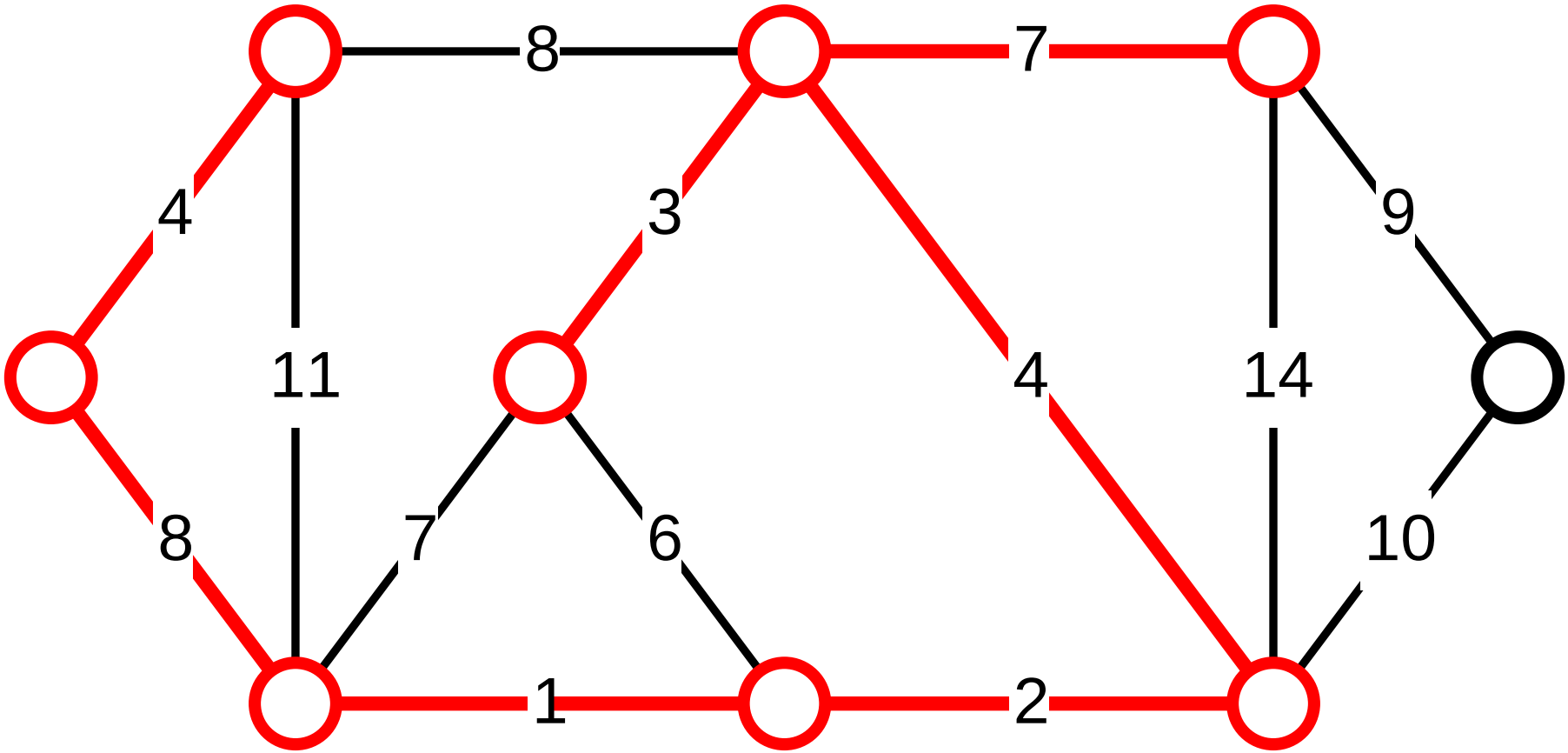
Algorytm Kruskala



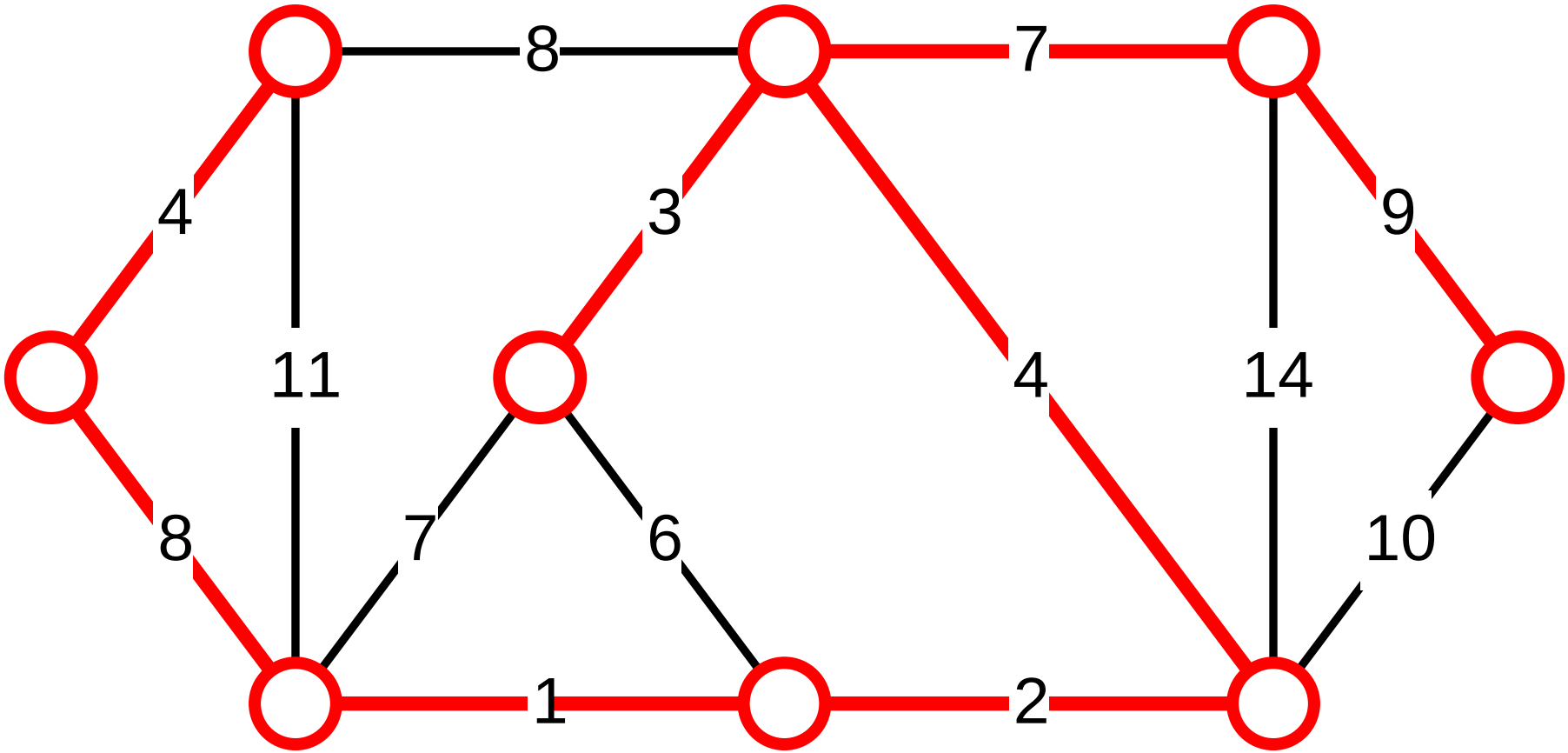
Algorytm Kruskala



Algorytm Kruskala



Algorytm Kruskala



Maksymalny przepływ w grafie

- etykiety krawędzi określają ich maksymalne przepustowości
- szukamy jaką ilość jesteśmy w stanie przez dany graf przesłać z wierzchołka startowego do wierzchołka docelowego
- w źródle odkręcamy kran
- etykiety – średnice rur
- maksymalny przepływ – jak szybko będzie leciała woda z ujścia

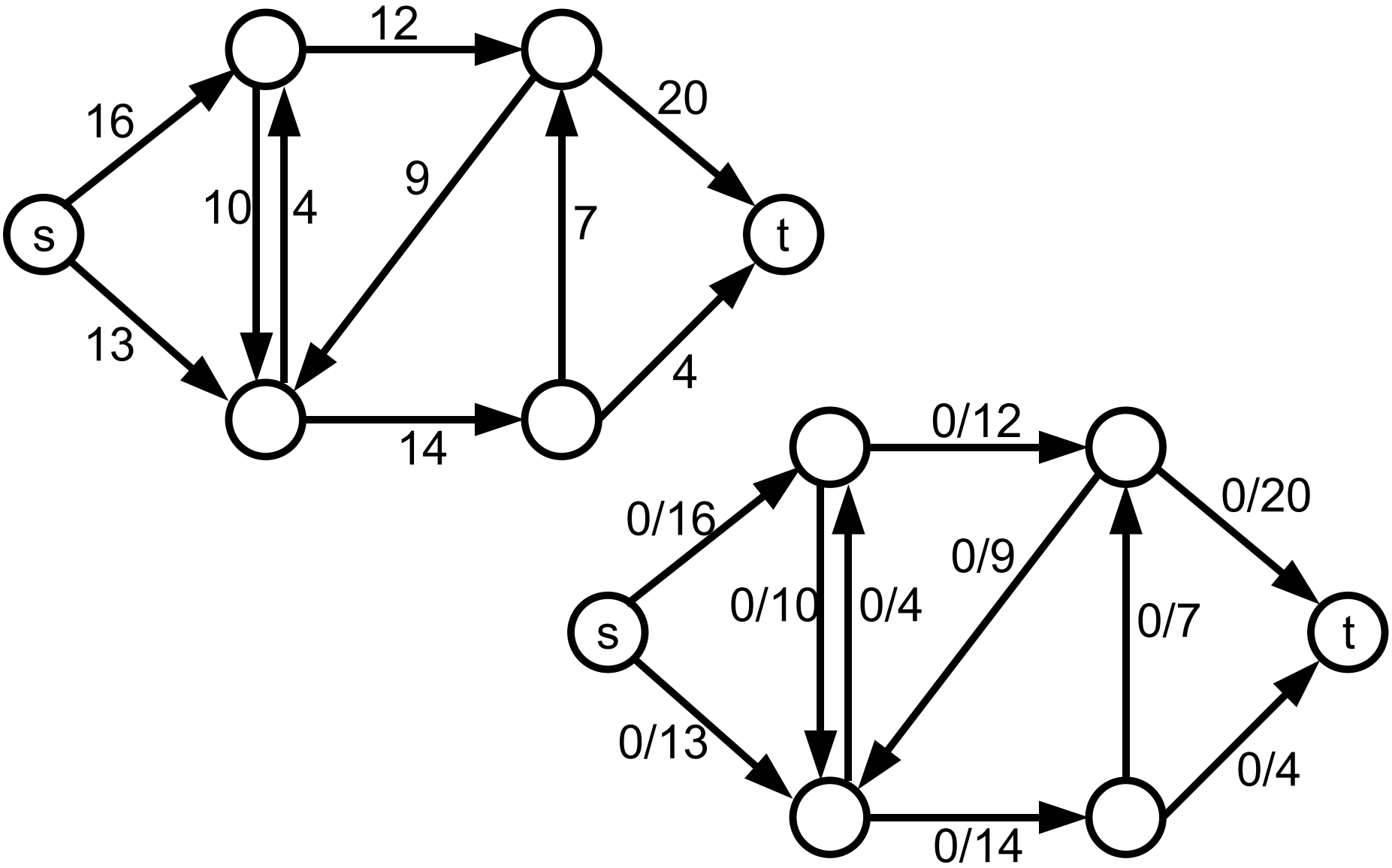
Algorytm Forda-Fulkersona

- zamień graf weściowy G na skierowany graf wyjściowy G', w którym początkowe wagi są równe przepustowości krawędzi
- graf G' opisuje niewykorzystaną przepustowość każdej krawędzi
- znajdź dowolną ścieżkę ze źródła do uścia przez krawędzie o dodatnich wagach
- pomniejsz wagi na krawędziach ze ścieżki o wartość najmniejszej etykiety ze ścieżki
- powtarzaj, dopóki można znaleźć nową ścieżkę

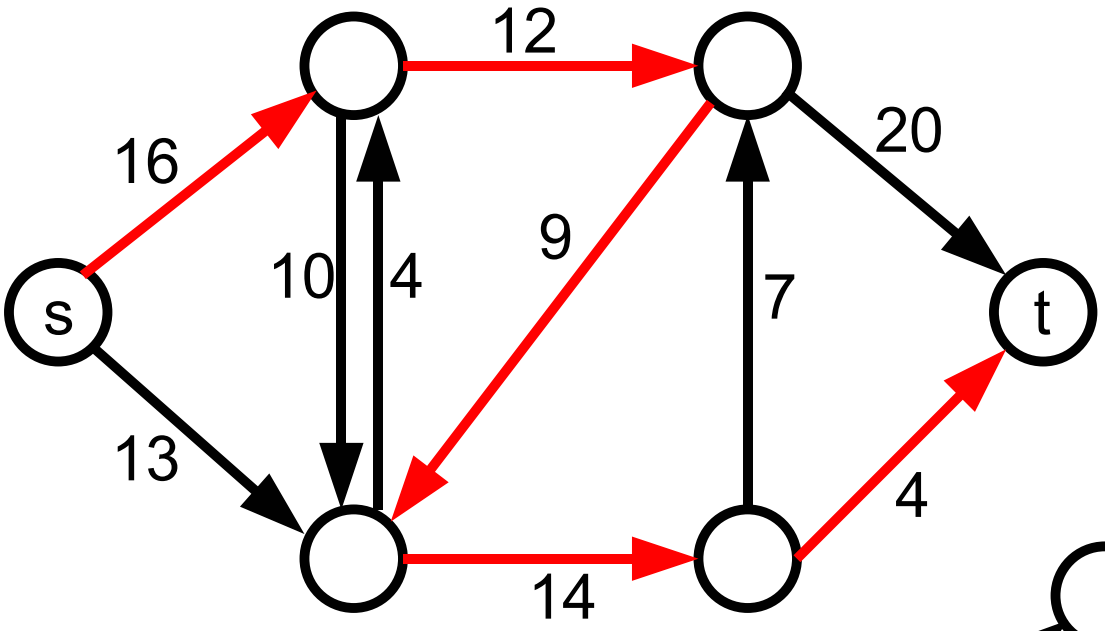
Algorytm Forda-Fulkersona

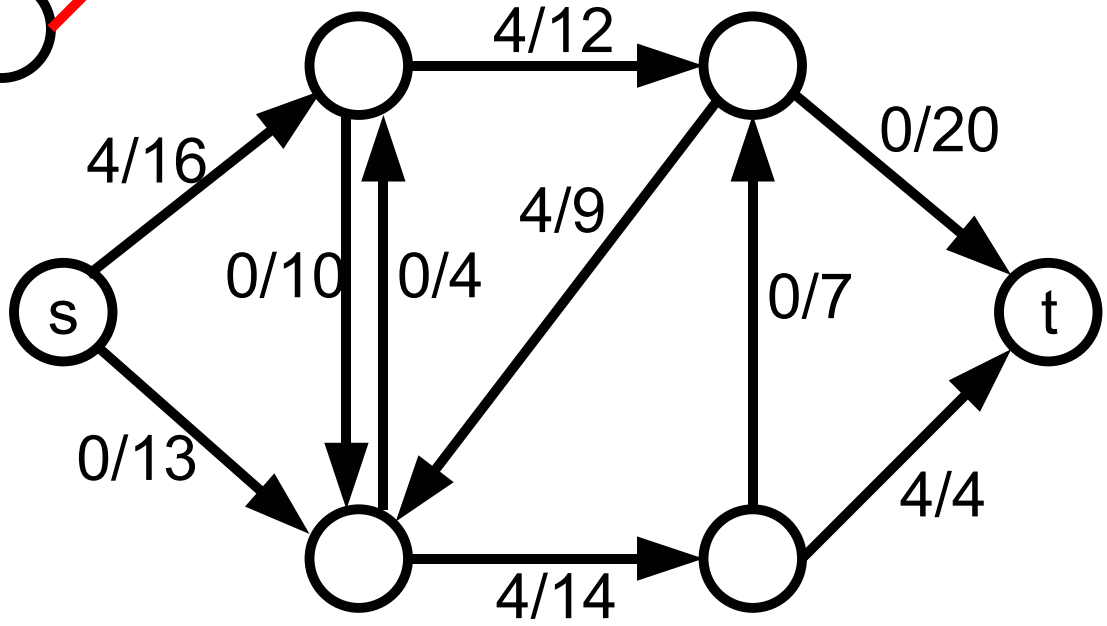
przekształć graf wejściowy G na graf skierowany G'
while istnieje ścieżka z s do t **do**
    d = najmniejsza przepustowość na ścieżce z s do t
    dla każdej pary u, v sąsiednich wierzchołków na ścieżce z s do t
    zmniejsz przepustowość w kierunku t o d, zwiększ przepustowość w kierunku s o d
end while

Algorytm Forda-Fulkersona

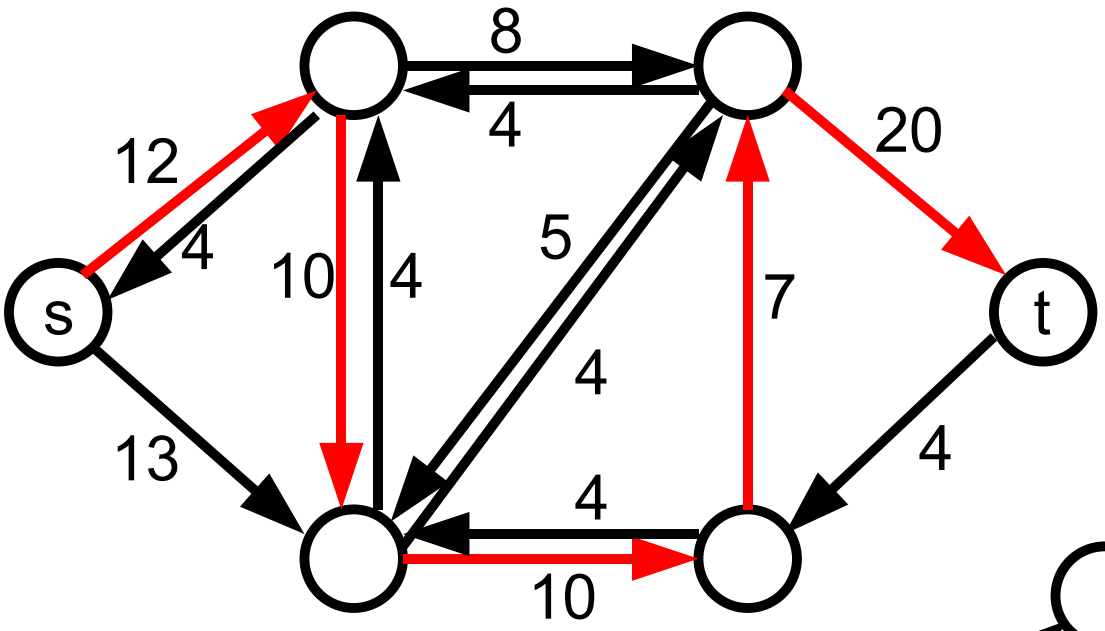


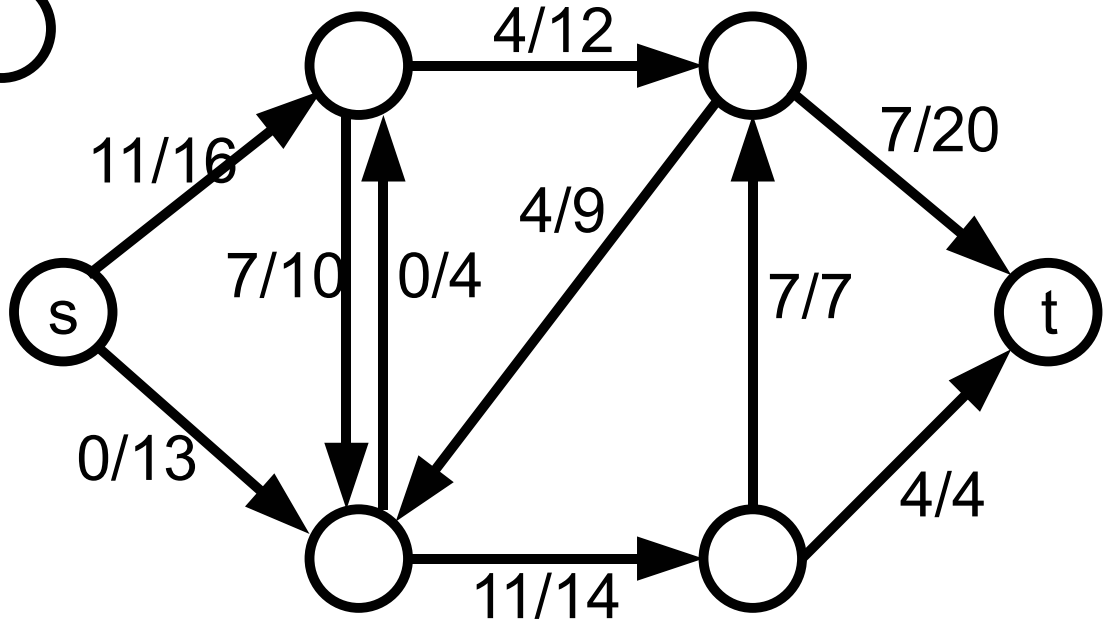
Algorytm Forda-Fulkersona



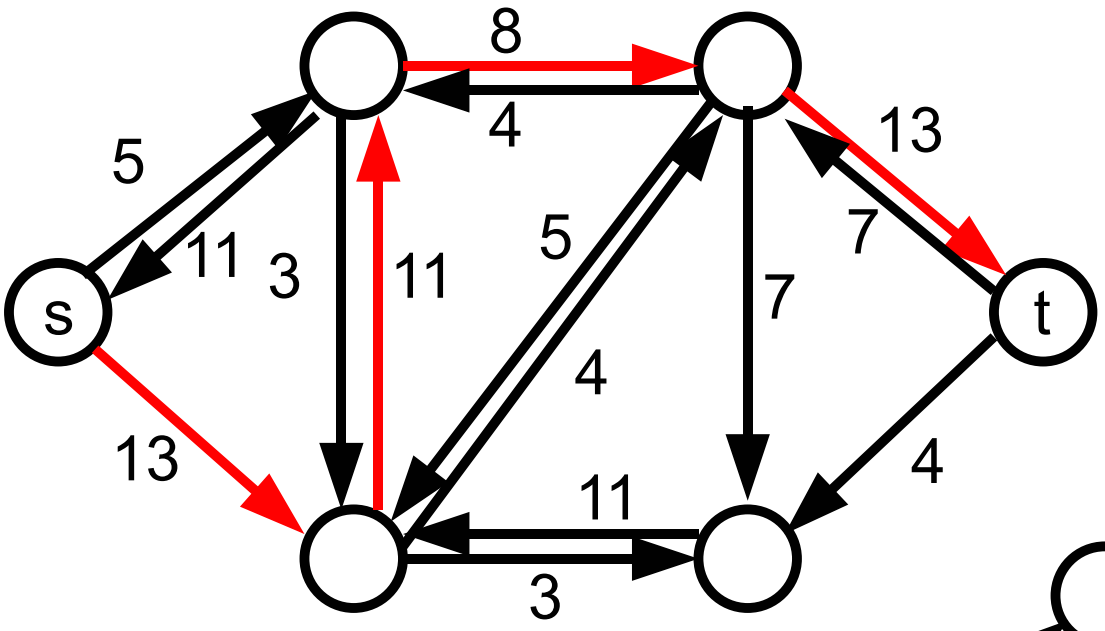


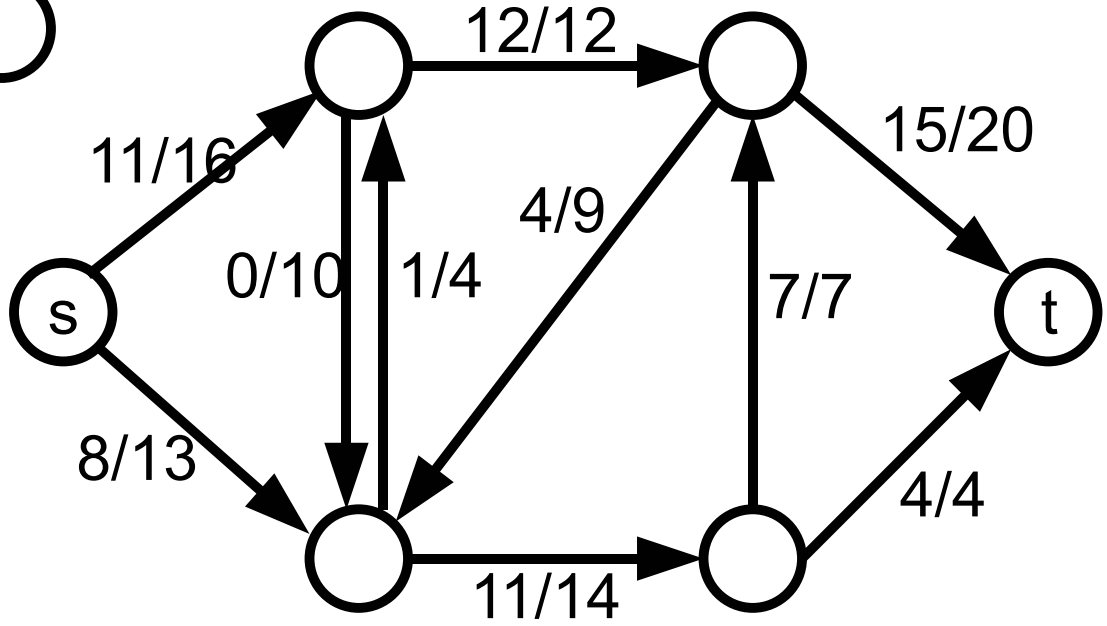
Algorytm Forda-Fulkersona



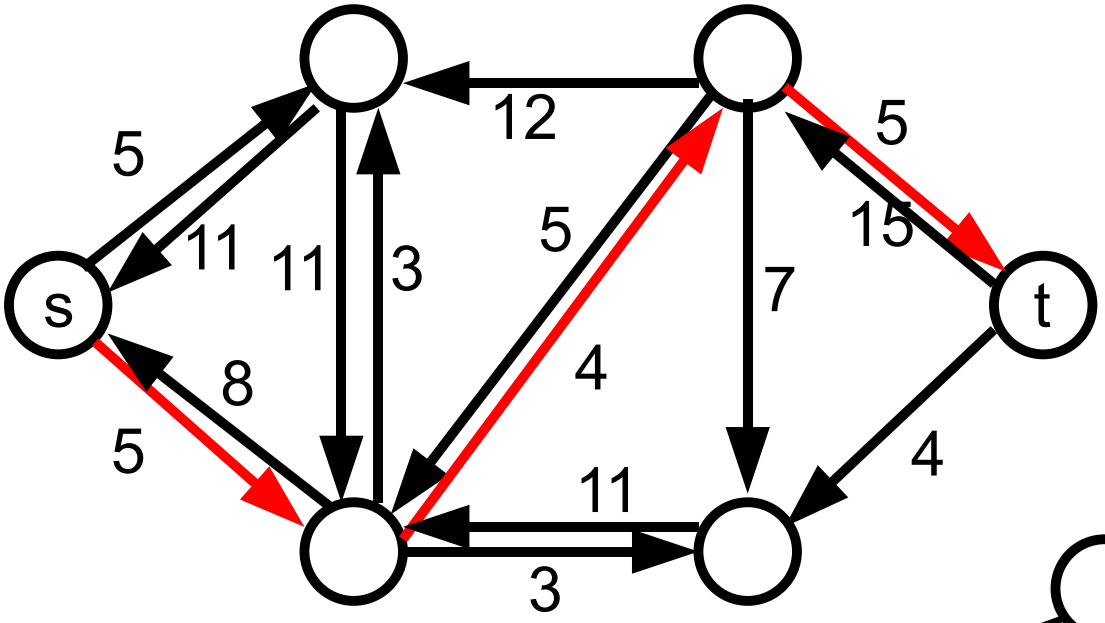


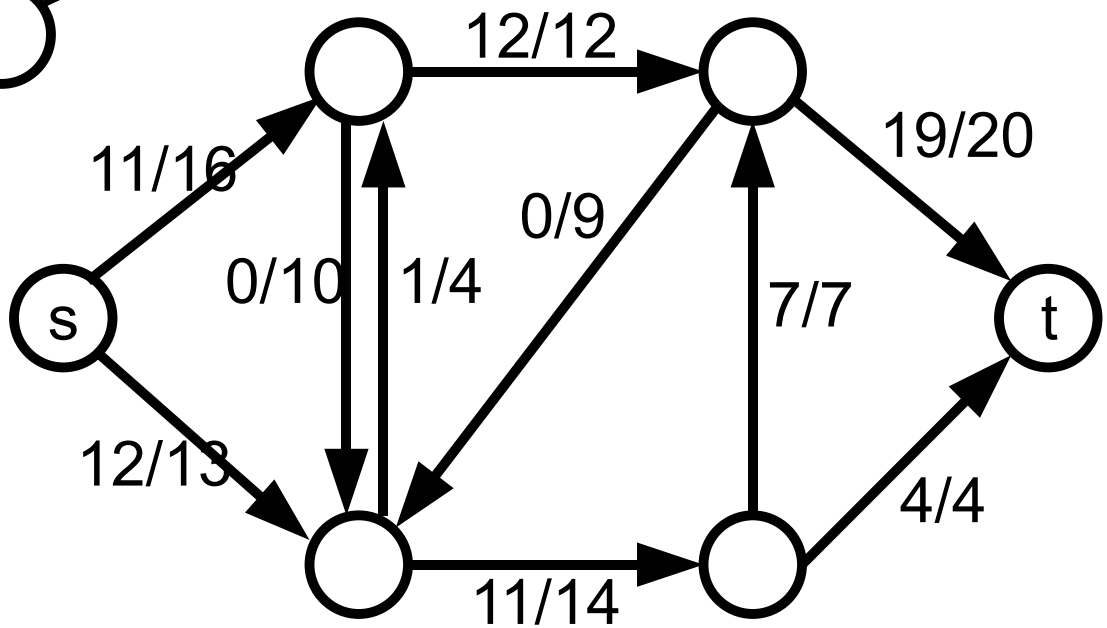
Algorytm Forda-Fulkersona



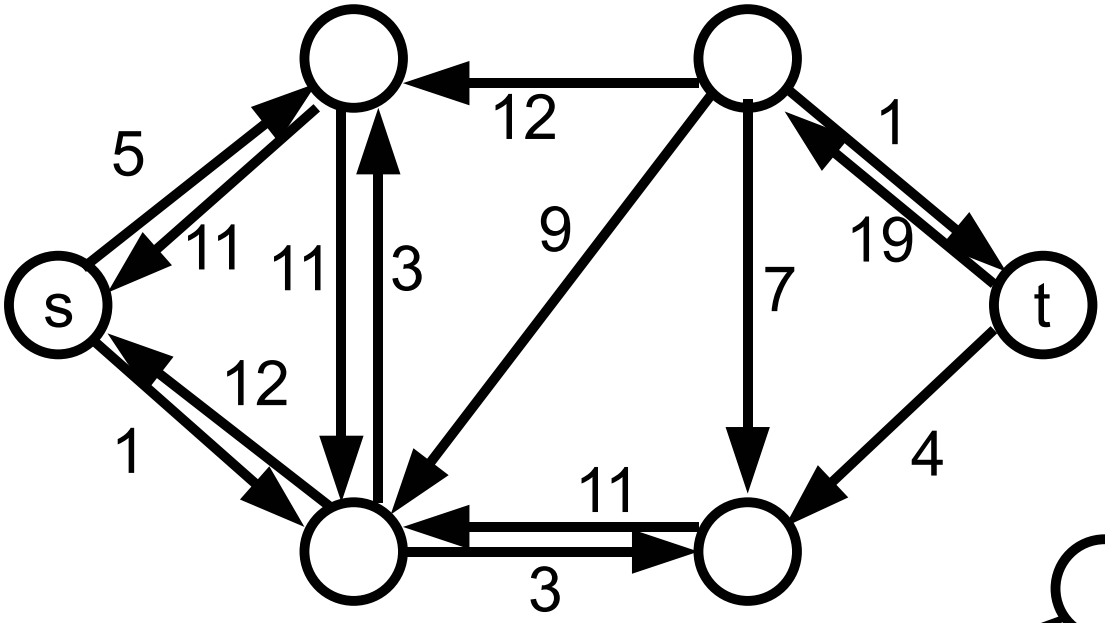


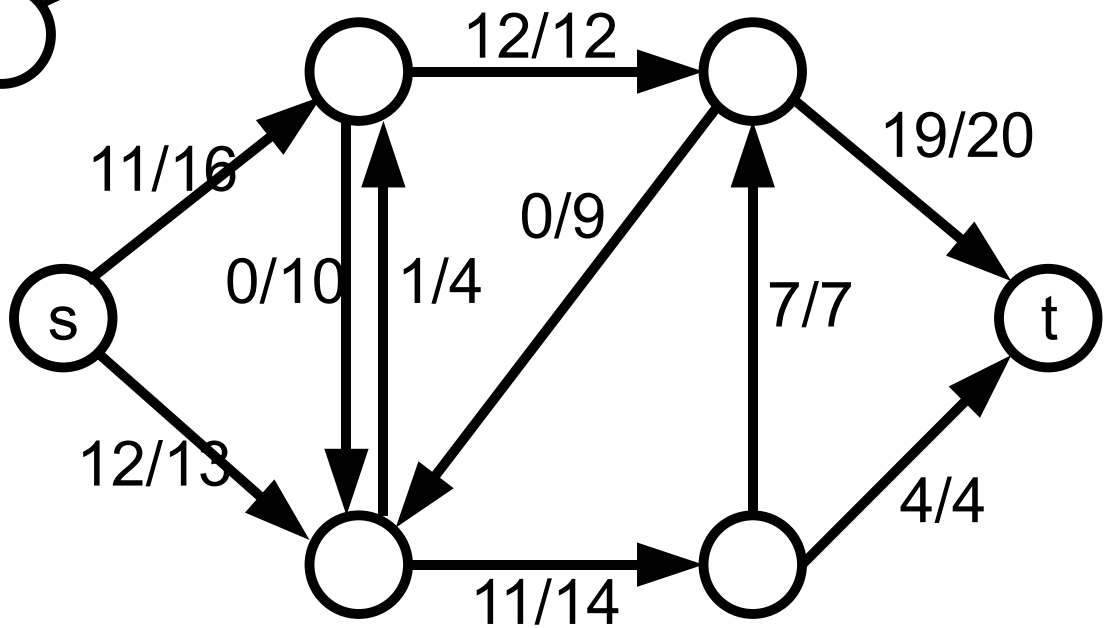
Algorytm Forda-Fulkersona





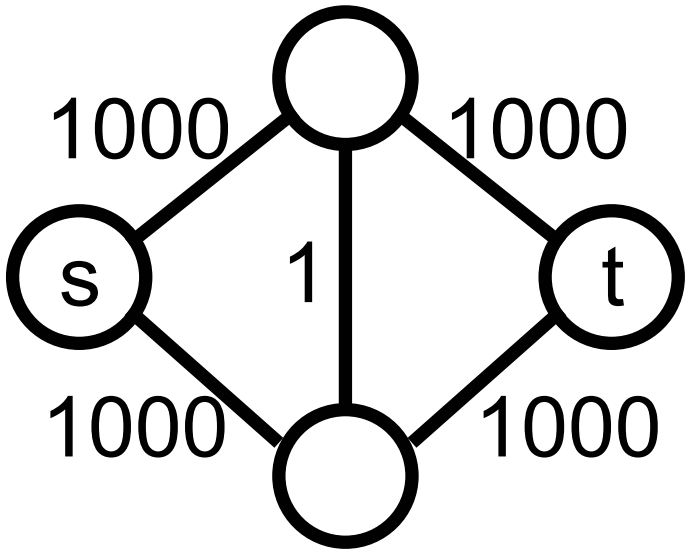
Algorytm Forda-Fulkersona





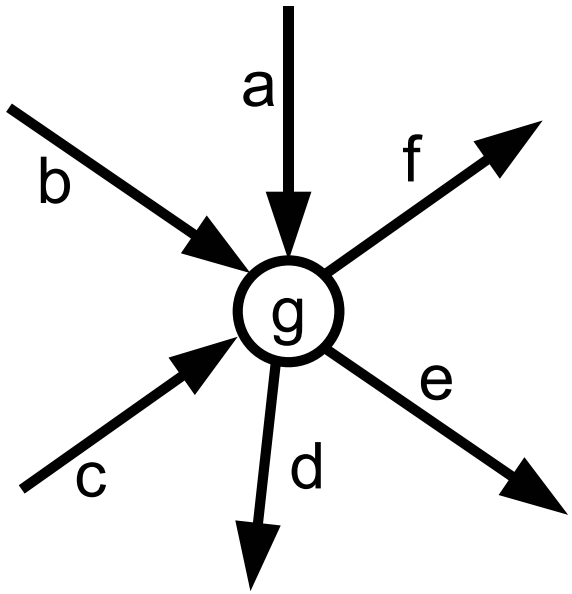
Algorytm Edmondsa-Karpa

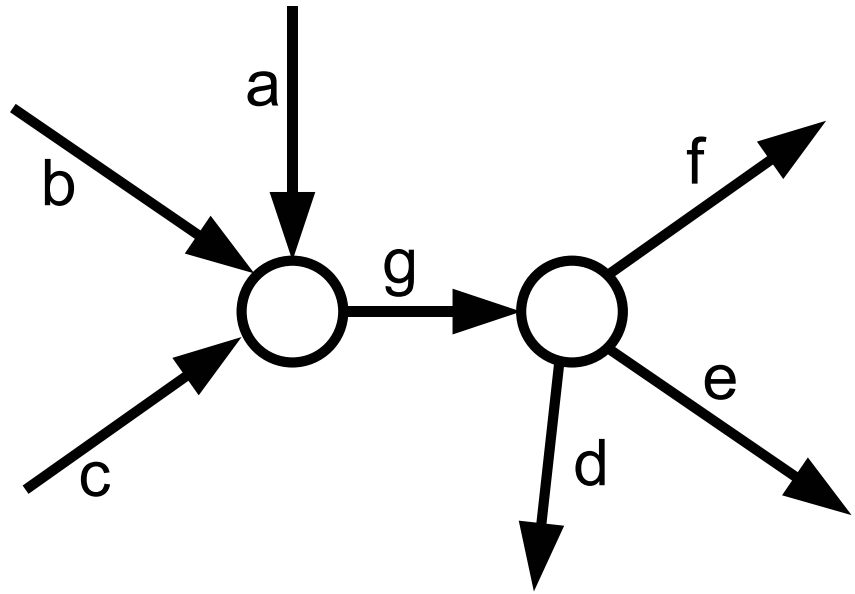
- wyszukujemy nie dowolnych, lecz najkrótszych ścieżek, przy użyciu przeszukiwania wszerz
- długość – liczba krawędzi



Maksymalny przepływ w grafie

- co jeżeli musimy uwzględnić także przepustowość wierzchołków?
- każdy wierzchołek zastępujemy krawędzią o przpustowości tego wierzchołka





Maksymalny przepływ w grafie

- co jeżeli mamy wiele źródeł i ujść?
- dodajemy nowy wierzchołek (nowe źródło) i łączymy je z wszystkimi źródłami krawędziami o dostatecznie dużj przepustowości
- dodajemy nowy wierzchołek (nowe ujście) i łączymy je z wszystkimi ujściami krawędziami o dostatecznie dużj przepustowości

Maksymalny przepływ w grafie