Koło informatyczne Władysława IV

Stanisław Bitner

listopad 2022

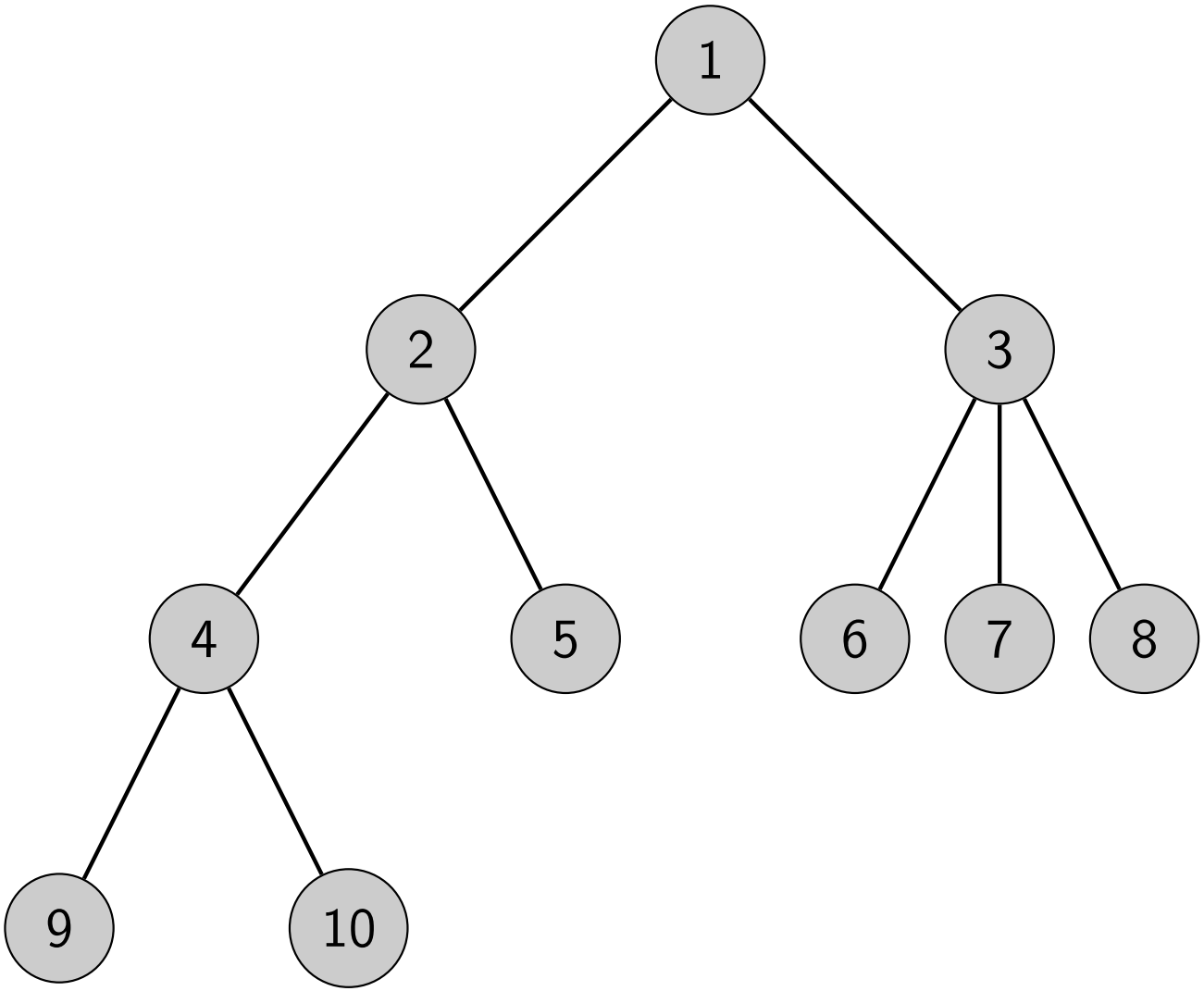
Spis treści

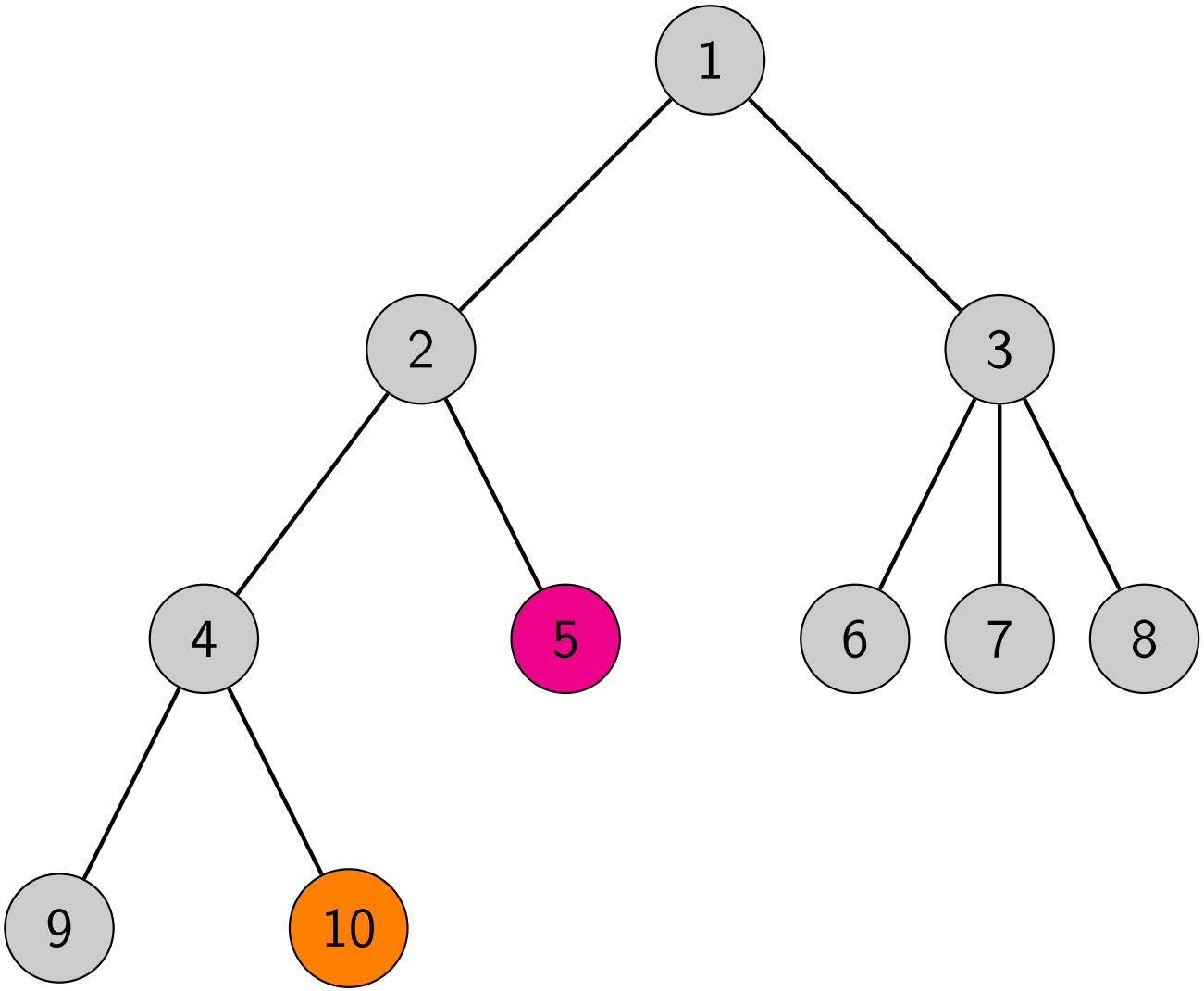
1. Najniższy wspólny przodek (LCA)
2. Minimalne drzewo rozpinające (MST)
3. Struktura zbiorów rozłącznych (DSU)
4. Sortowanie topologiczne (Toposort)
5. Silnie spójne składowe (SCC)
6. Najlżejsze (najkrótsze) ścieżki
7. Parametr low (mosty i punkty artykulacji)
8. Drzewa przedziałowe

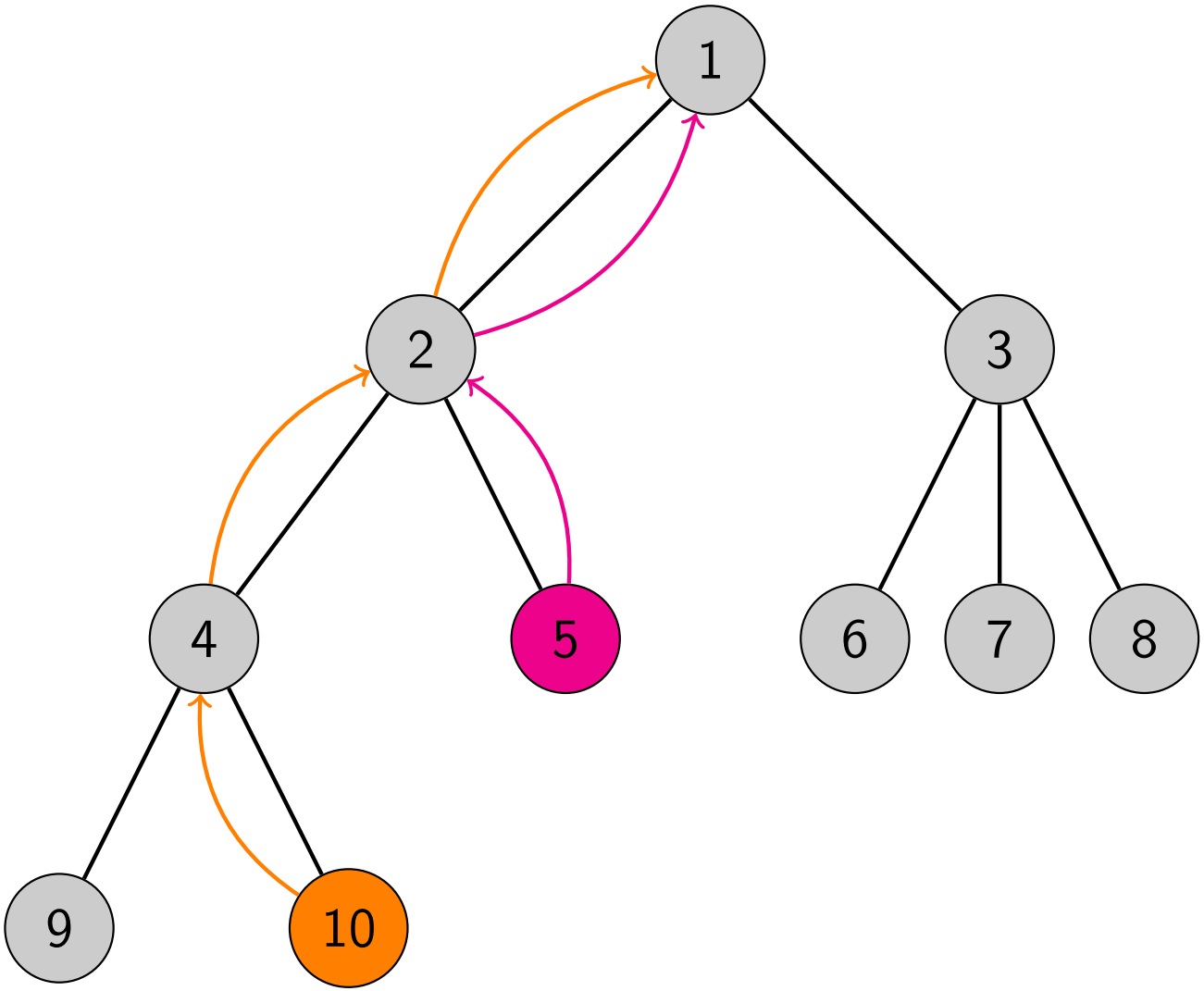
Najniższy wspólny przodek (LCA)

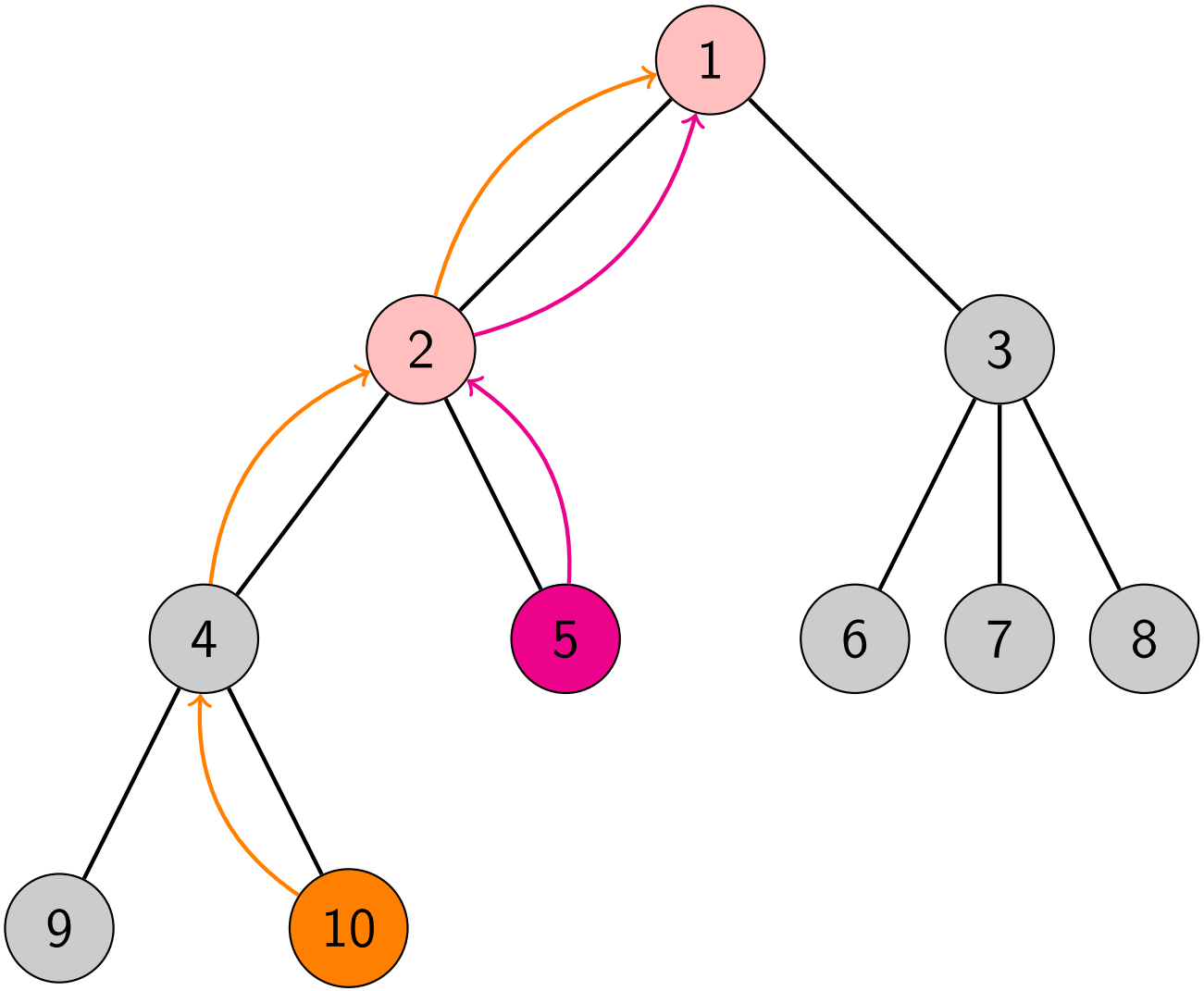
- Wprowadzenie
- Naiwne podejście
- Binary lifting

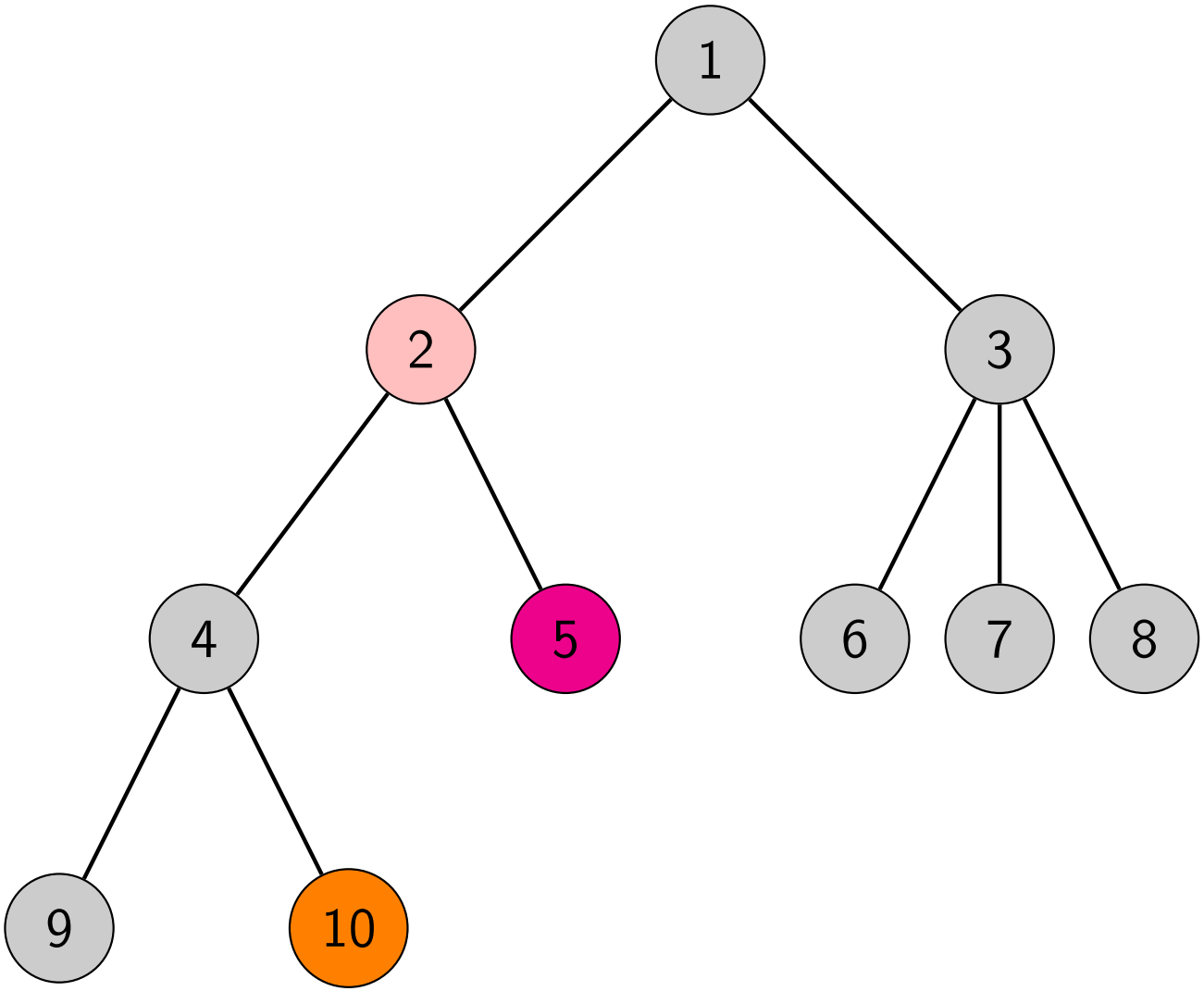
Niech G będzie drzewem.
Dla każdego zapytania typu (v, u) chcemy znaleźć najniższego wspólnego przodka wierzchołka v i wierzchołka u, czyli taki wierzchołek w, że należy on do ścieżki z v do korzenia i ścieżki z u do korzenia. Jeśli istnieje wiele takich wierzchołków, to chcemy wyznaczyć ten najodleglejszy od korzenia drzewa.


1
2
3
4
5
6
7
8
9
10


1
2
3
4
5
6
7
8
9
10


1
2
3
4
5
6
7
8
9
10


1
2
3
4
5
6
7
8
9
10


1
2
3
4
5
6
7
8
9
10

Naiwne podejście

Oczywistym rozwiązaniem byłoby skakanie do ojca z niższego z wierzchołków v i u.

Naiwne podejście

Oczywistym rozwiązaniem byłoby skakanie do ojca z niższego z wierzchołków v i u.

```
while (v != u) {
  if (glebokosc(v) > glebokosc(u))
    v = ojciec[v];
  else
    u = ojciec[u];
}
return v;
```

Naiwne podejście

Oczywistym rozwiązaniem byłoby skakanie do ojca z niższego z wierzchołków v i u.

```
while (v != u) {
  if (glebokosc(v) > glebokosc(u))
    v = ojciec[v];
  else
    u = ojciec[u];
}
return v;
```

Takie rozwiązanie jest jednak zbyt wolne i w pesymistycznym przypadku wykonuje $O(n)$ operacji.

Naiwne podejście

Nasz naiwny algorytm można z łatwością usprawnić.

Naiwne podejście

Nasz naiwny algorytm można z łatwością usprawnić.
Nie musimy skakać dwoma wierzchołkami. Wystarczy, że będziemy skakać jednym z nich, dopóki nie doskoczymy do przodka drugiego wierzchołka.

Naiwne podejście

Nasz naiwny algorytm można z łatwością usprawnić.
Nie musimy skakać dwoma wierzchołkami. Wystarczy, że będziemy skakać jednym z nich, dopóki nie doskoczymy do przodka drugiego wierzchołka.

```
while (!jest_przodkiem(v, u))
  v = ojciec[v];
return v;
```

Binary lifting

Jedyną wadą poprzedniego algorytmu było to, że skakaliśmy za każdym razem o tylko 1 wierzchołek w górę.

Binary lifting

Jedyną wadą poprzedniego algorytmu było to, że skakaliśmy za każdym razem o tylko 1 wierzchołek w górę. Aby pozbyć się tego problemu, możemy skakać o kilka – na przykład 2^k – wierzchołków w górę.

Binary lifting

Jedyną wadą poprzedniego algorytmu było to, że skakaliśmy za każdym razem o tylko 1 wierzchołek w górę. Aby pozbyć się tego problemu, możemy skakać o kilka – na przykład 2^k – wierzchołków w górę.

```
while (!jest_przodkiem(v, u))
  v = skocz_o_kilka(v);
return v;
```

Binary lifting

Jedyną wadą poprzedniego algorytmu było to, że skakaliśmy za każdym razem o tylko 1 wierzchołek w górę. Aby pozbyć się tego problemu, możemy skakać o kilka – na przykład 2^k – wierzchołków w górę.

```
while (!jest_przodkiem(v, u))
  v = skocz_o_kilka(v);
return v;
```

Tutaj pojawiają się 2 problemy:

Binary lifting

Jedyną wadą poprzedniego algorytmu było to, że skakaliśmy za każdym razem o tylko 1 wierzchołek w górę. Aby pozbyć się tego problemu, możemy skakać o kilka – na przykład 2^k – wierzchołków w górę.

```
while (!jest_przodkiem(v, u))
  v = skocz_o_kilka(v);
return v;
```

Tutaj pojawiają się 2 problemy:

1. nie mamy zdefiniowanej funkcji do skakania;

Binary lifting

Jedyną wadą poprzedniego algorytmu było to, że skakaliśmy za każdym razem o tylko 1 wierzchołek w górę. Aby pozbyć się tego problemu, możemy skakać o kilka – na przykład 2^k – wierzchołków w górę.

```
while (!jest_przodkiem(v, u))
  v = skocz_o_kilka(v);
return v;
```

Tutaj pojawiają się 2 problemy:

1. nie mamy zdefiniowanej funkcji do skakania;
2. nie kontrolujemy długości skoku (możemy skoczyć za daleko).

Binary lifting

Jako jedno z rozwiązań uprzednio wspomnianych poprzednio problemów proponuję takie rozwiązanie. Skaczemy wierzchołkiem v do najniższego wierzchołka niebędącego przodkiem u.
Ojcem takiego wierzchołka będzie $LCA(v, u)$.
Jeśli chodzi o długość skoków, to będziemy skakać o coraz mniejsze potęgi dwójki – można w ten sposób uzyskać dowolną liczbę.

Binary lifting

Jako jedno z rozwiązań uprzednio wspomnianych poprzednio problemów proponuję takie rozwiązanie. Skaczemy wierzchołkiem v do najniższego wierzchołka niebędącego przodkiem u.
Ojcem takiego wierzchołka będzie $LCA(v, u)$.
Jeśli chodzi o długość skoków, to będziemy skakać o coraz mniejsze potęgi dwójki – można w ten sposób uzyskać dowolną liczbę.

```
for (int skok = LOG_N; --skok >= 0;)
  if (!jest_przodkiem(ojciec[v][skok], u))
    v = ojciec[v][skok];
return ojciec[v][0];
```

Binary lifting

Pozostaje nam wyznaczać szybko wartości funkcji `ojciec`.

Binary lifting

Pozostaje nam wyznaczać szybko wartości funkcji `ojciec`.Można łatwo ją spreprocesować wykonując `dfs`-a z korzenia drzewa.

```
int ojciec[MAX_N][LOG_N];
void dfs(int v, int p) {
  visited[v] = true;
  ojciec[v][0] = p;
  for (int skok = 1; skok < LOG_N; skok++) {
    int u = ojciec[v][skok - 1];
    ojciec[v][skok] = ojciec[u][skok - 1];
  }
  for (auto u : G[v])
    if (!visited[u])
      dfs(u, v);
}
```

Binary lifting

Pozostaje nam wyznaczać szybko wartości funkcji `ojciec`.Można łatwo ją spreprocesować wykonując `dfs`-a z korzenia drzewa.

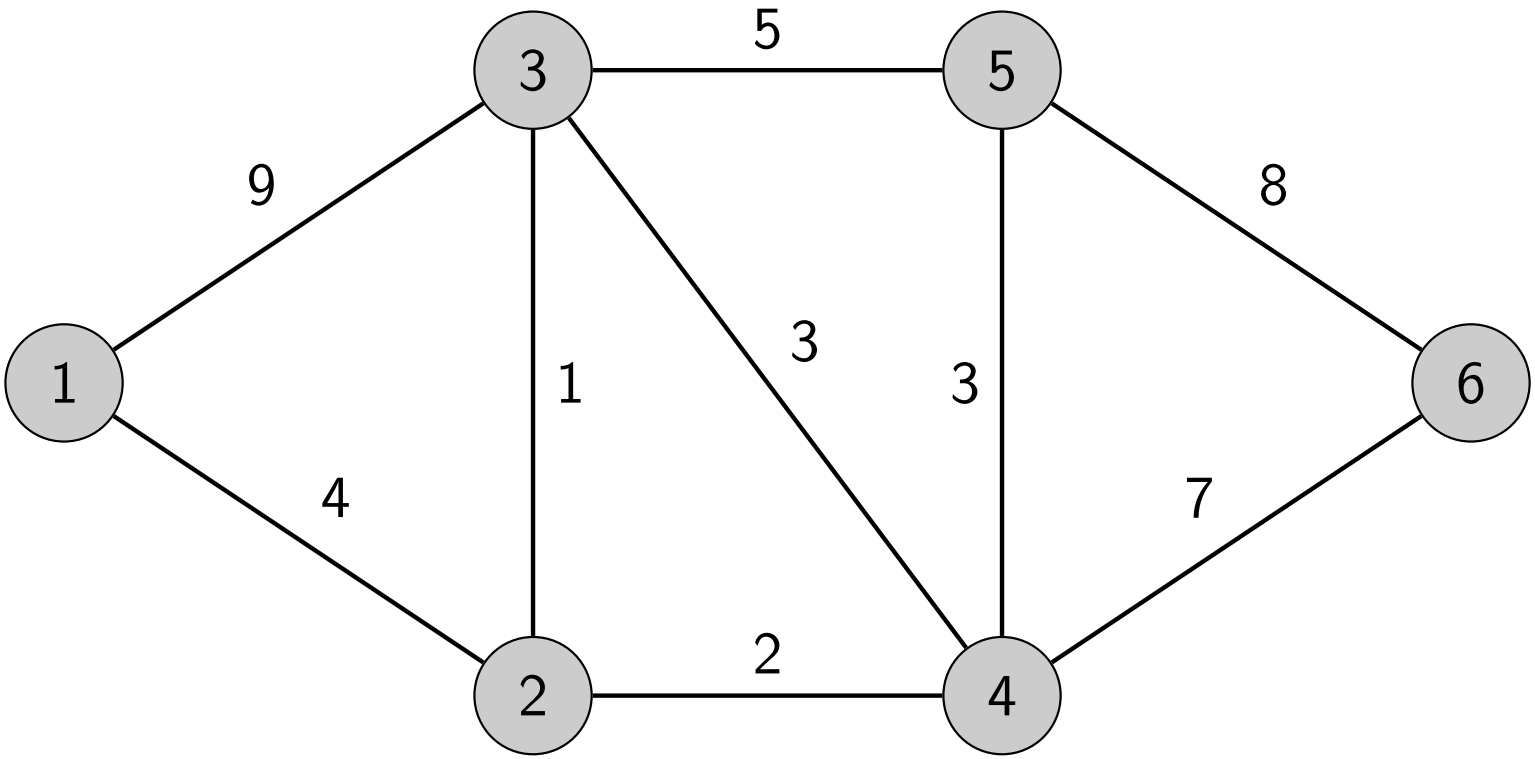
```
int ojciec[MAX_N][LOG_N];
void dfs(int v, int p) {
  visited[v] = true;
  ojciec[v][0] = p;
  for (int skok = 1; skok < LOG_N; skok++) {
    int u = ojciec[v][skok - 1];
    ojciec[v][skok] = ojciec[u][skok - 1];
  }
  for (auto u : G[v])
    if (!visited[u])
      dfs(u, v);
}
```

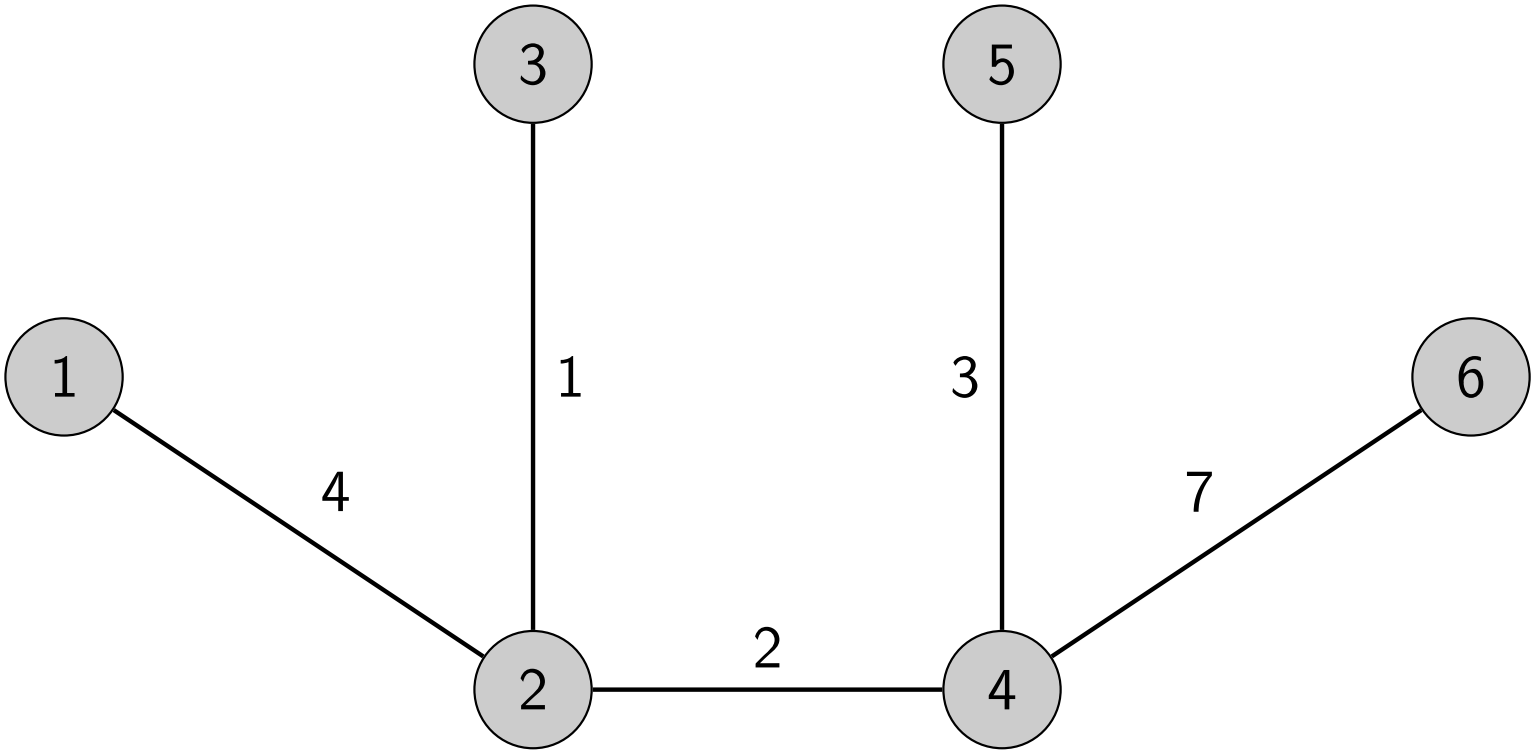
Takie rozwiązanie okazuje się już być wystarczająco szybkie i pozwala na odpowiadanie na zapytania w złożoności $O(\log n)$.

Minimalne drzewo rozpinające (MST)

- Wprowadzenie
- Algorytm Kruskala

Niech G będzie ważonym grafem nieskierowanym.
Chcemy znaleźć poddrzewo grafu G, które łączy wszystkie jego wierzchołki (czyli jego drzewo rozpinające) i ma najmniejszą wagę (sumę wszystkich wag krawędzi) ze wszystkich możliwych drzew rozpinających. Takie drzewo rozpinające nazywamy minimalnym drzewem rozpinającym (ang. *minimum spanning tree*).


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


3
5
1
1
3
6
4
7
2
2
4

- Jeżeli wszystkie wagi krawędzi są różne, to istnieje tylko jedno MST, w przeciwnym razie możliwe jest istnienie wielu.

- Jeżeli wszystkie wagi krawędzi są różne, to istnieje tylko jedno MST, w przeciwnym razie możliwe jest istnienie wielu.
- MST jest także drzewem rozpinającym o najmniejszym iloczynie wag krawędzi (prosty dowód poprzez zamianę wag na ich logarytmy).

- Jeżeli wszystkie wagi krawędzi są różne, to istnieje tylko jedno MST, w przeciwnym razie możliwe jest istnienie wielu.
- MST jest także drzewem rozpinającym o najmniejszym iloczynie wag krawędzi (prosty dowód poprzez zamianę wag na ich logarytmy).
- Największa waga krawędzi w MST jest najmniejsza spośród wszystkich maksymalnych wag krawędzi wszystkich możliwych drzew rozpinających.

- Jeżeli wszystkie wagi krawędzi są różne, to istnieje tylko jedno MST, w przeciwnym razie możliwe jest istnienie wielu.
- MST jest także drzewem rozpinającym o najmniejszym iloczynie wag krawędzi (prosty dowód poprzez zamianę wag na ich logarytmy).
- Największa waga krawędzi w MST jest najmniejsza spośród wszystkich maksymalnych wag krawędzi wszystkich możliwych drzew rozpinających.
- Maksymalne drzewo rozpinające może zostać uzyskane tak samo jak MST, wystarczy zamienić wagi krawędzi na ujemne.

Algorytm Kruskala

Twórcą algorytmu jest Joseph Bernard Kruskal. Algorytm jest bardzo prosty w działaniu:

Algorytm Kruskala

Twórcą algorytmu jest Joseph Bernard Kruskal. Algorytm jest bardzo prosty w działaniu:

1. sortujemy krawędzie niemalejąco według wag;

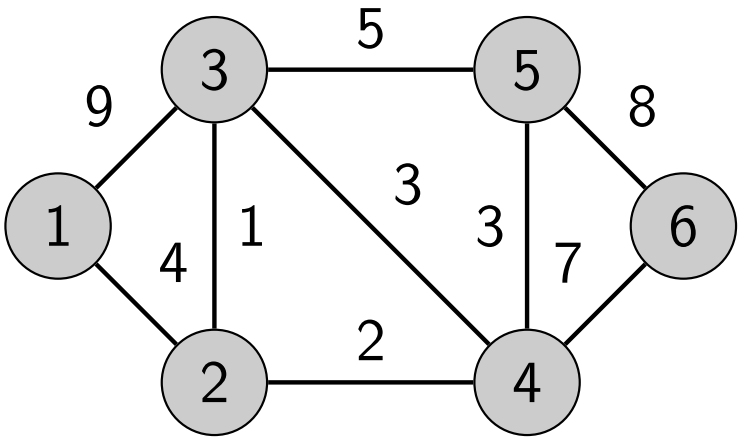
Algorytm Kruskala

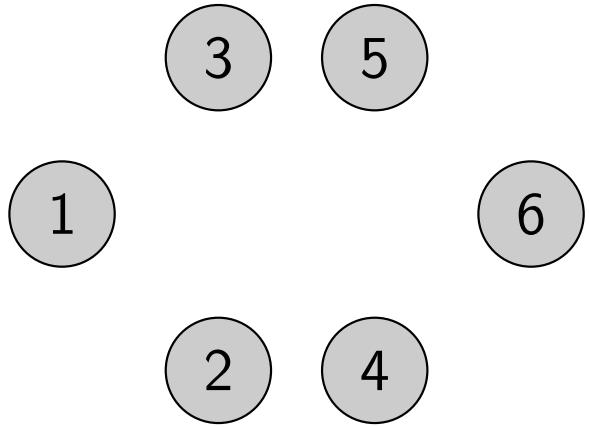
Twórcą algorytmu jest Joseph Bernard Kruskal. Algorytm jest bardzo prosty w działaniu:

1. sortujemy krawędzie niemalejąco według wag;
2. przeglądamy krawędzie od najlżejszych i zachłannie dodajemy do wyniku te, które łączą dwa do tej pory niepołączone wierzchołki.

Algorytm Kruskala

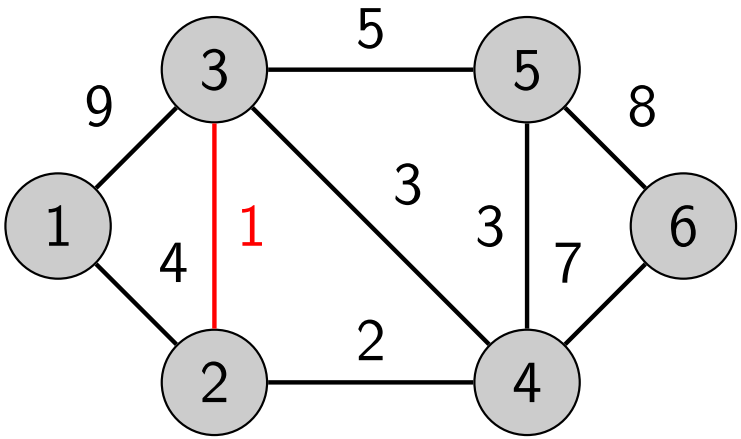
Przykład działania

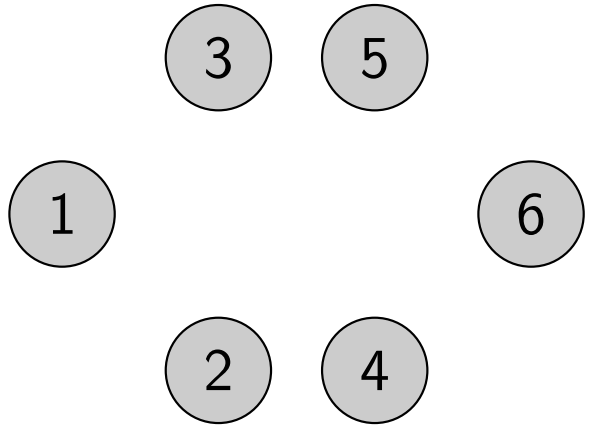




Algorytm Kruskala

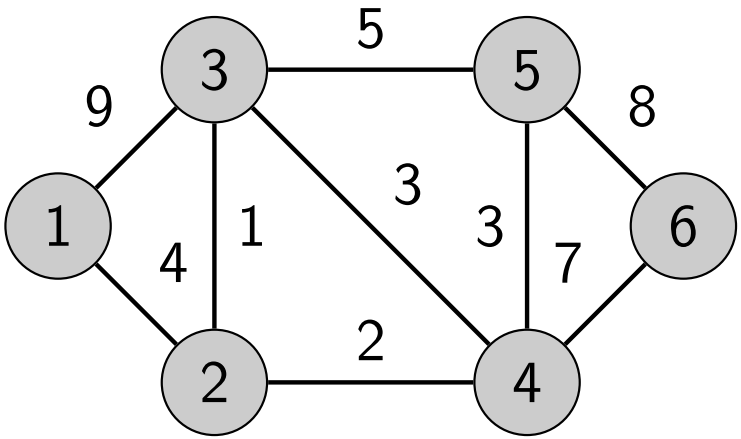
Przykład działania

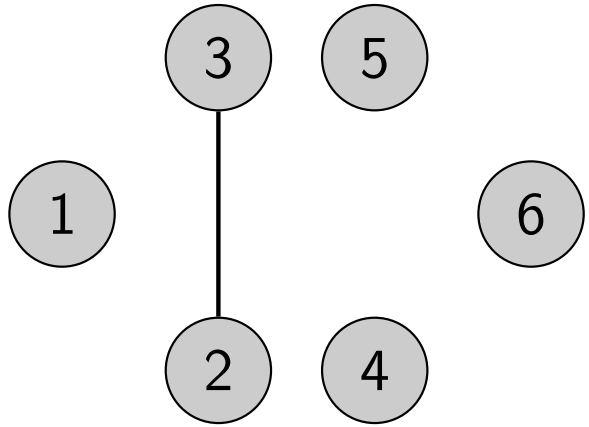




Algorytm Kruskala

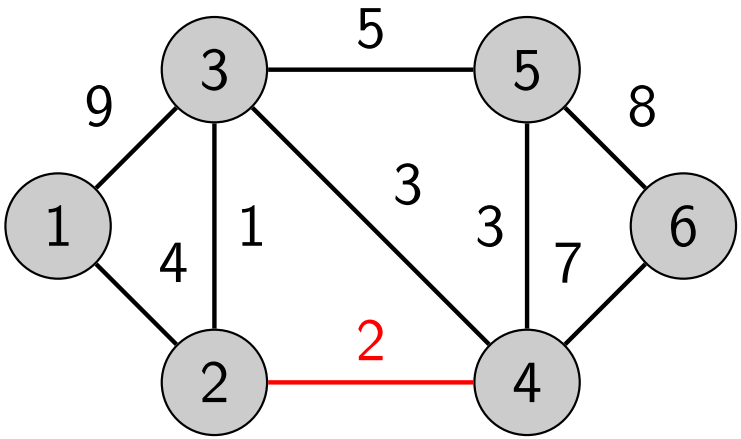
Przykład działania

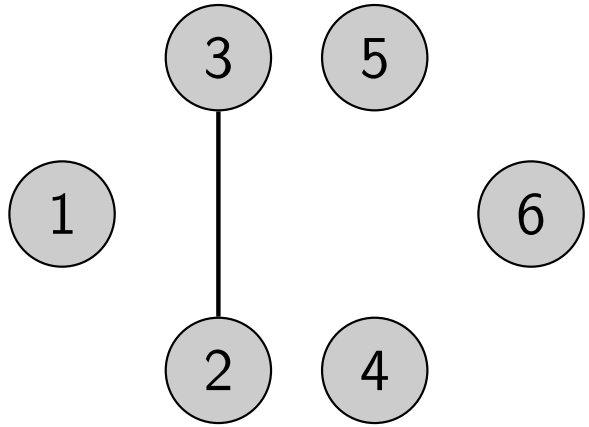




Algorytm Kruskala

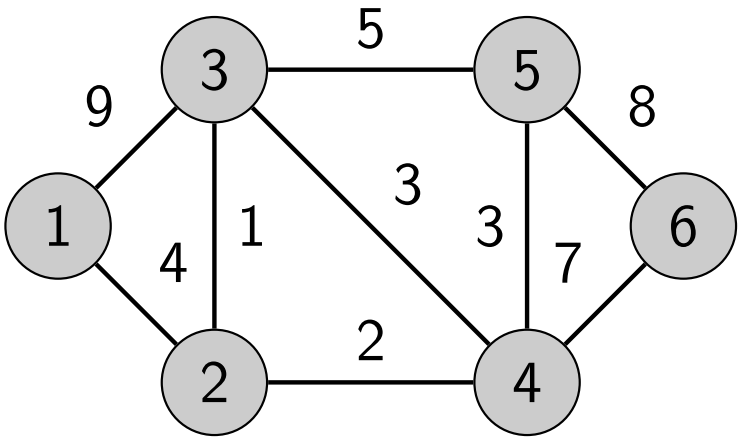
Przykład działania

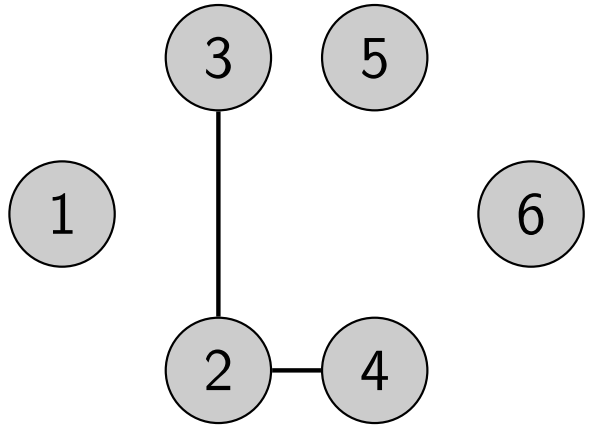




Algorytm Kruskala

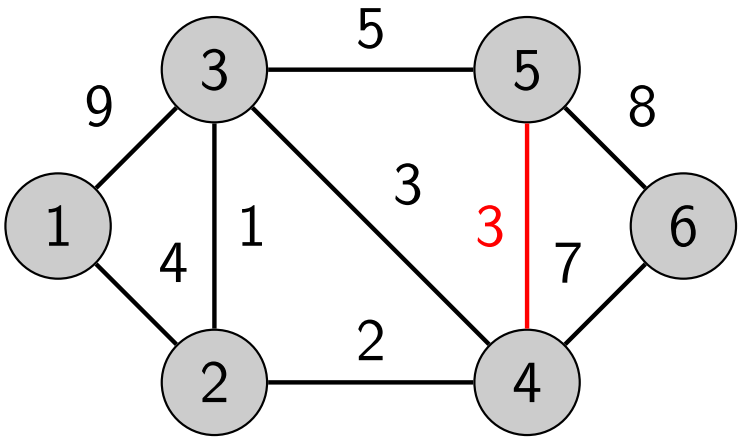
Przykład działania

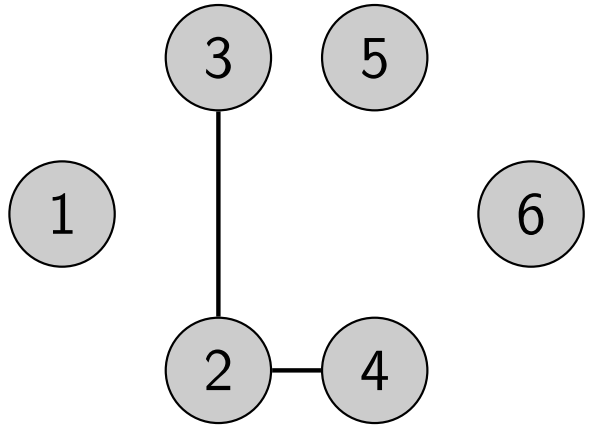




Algorytm Kruskala

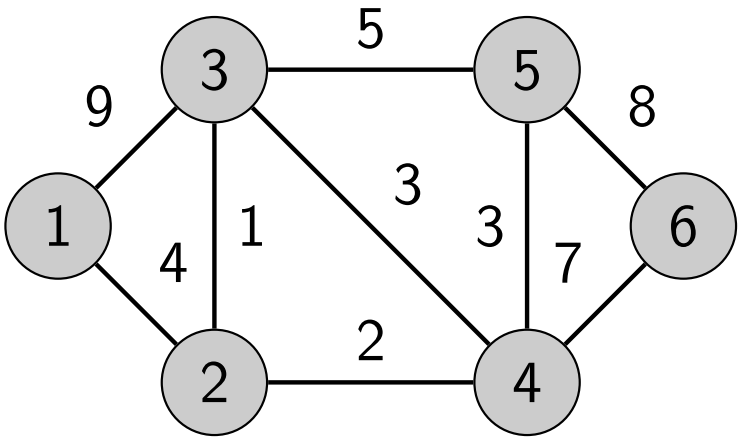
Przykład działania

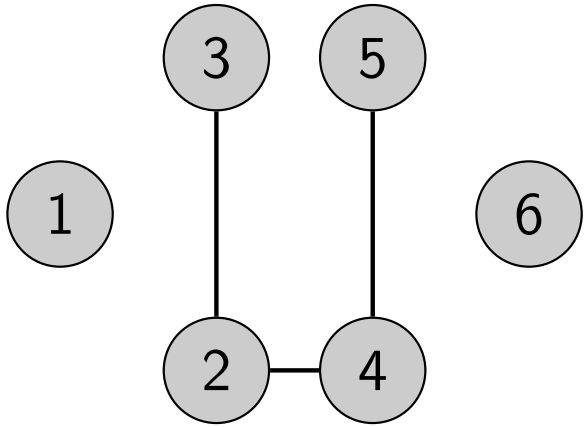




Algorytm Kruskala

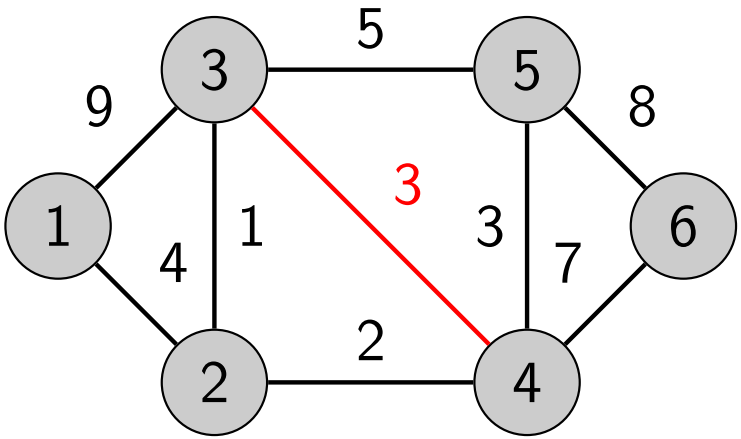
Przykład działania

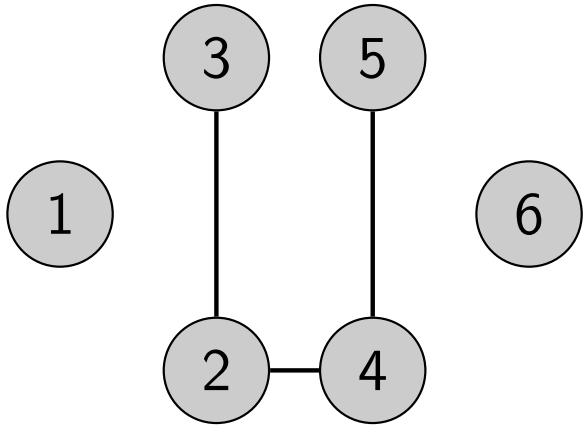




Algorytm Kruskala

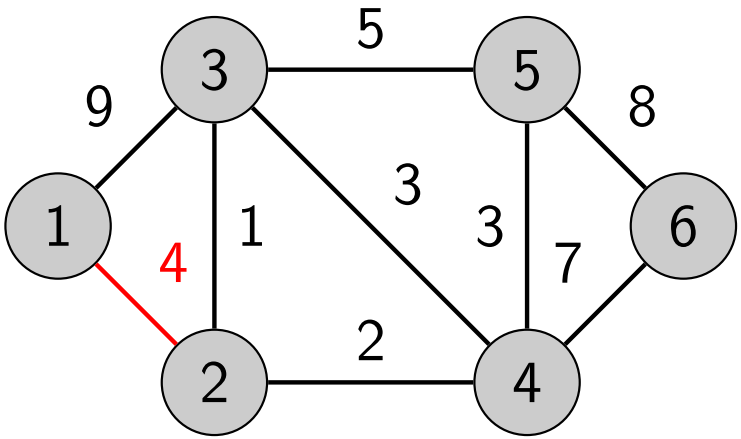
Przykład działania

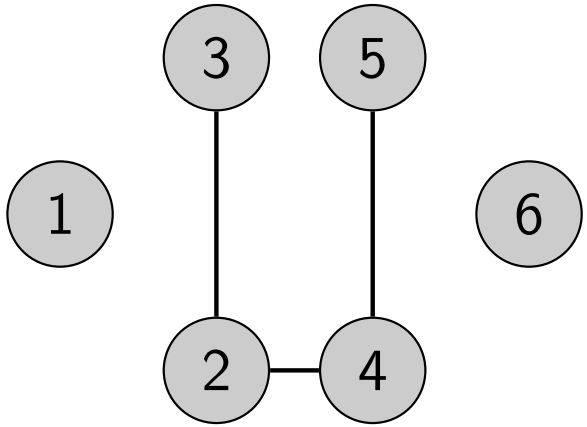


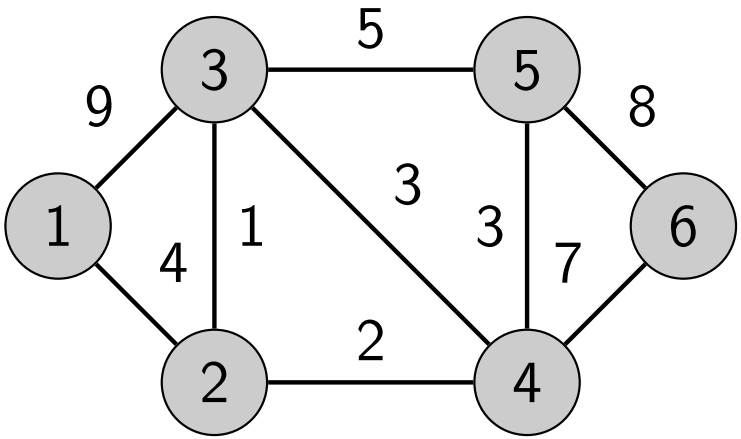


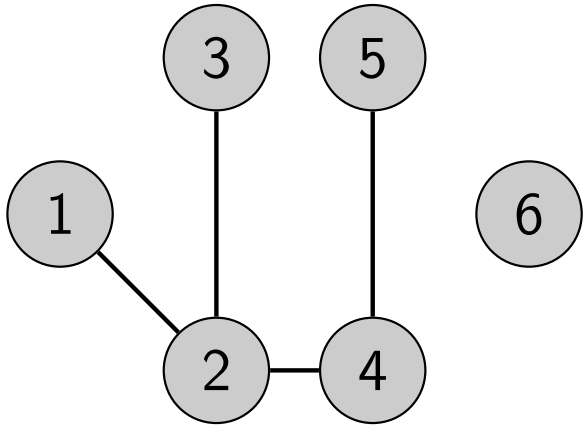
Algorytm Kruskala

Przykład działania



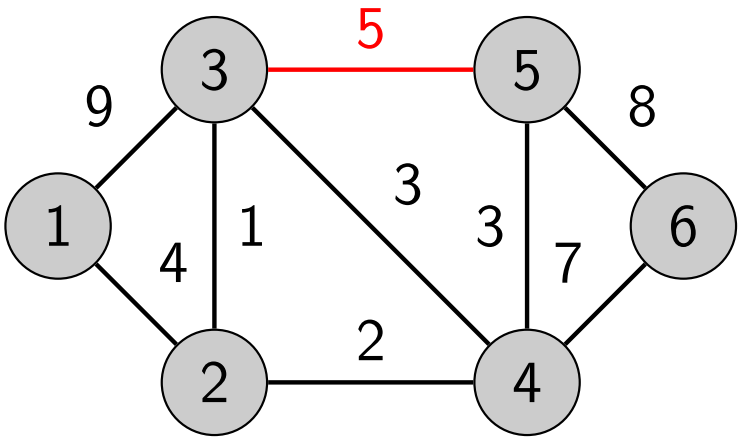


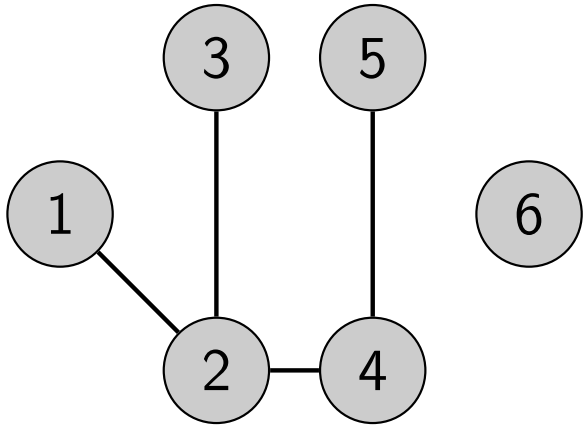

1
2
3
4
5
6
9
5
8
4
1
3
3
7
2

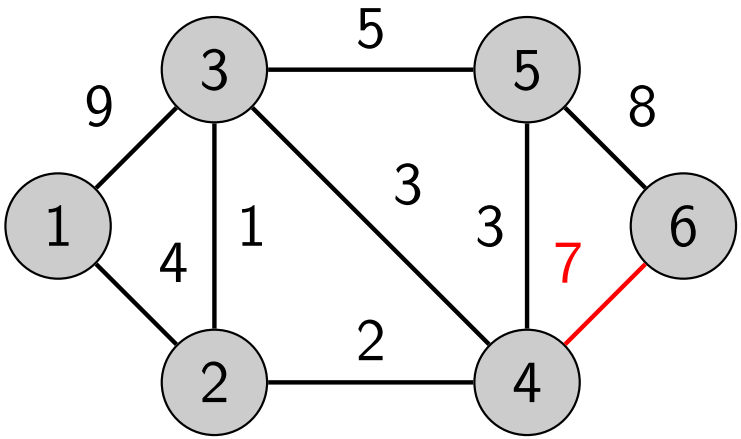

1
2
3
4
5
6

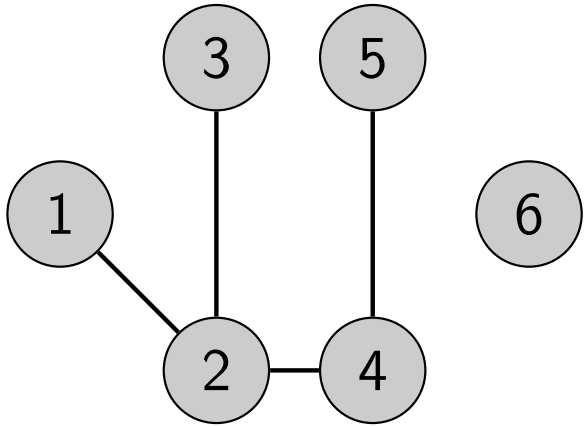
Algorytm Kruskala

Przykład działania



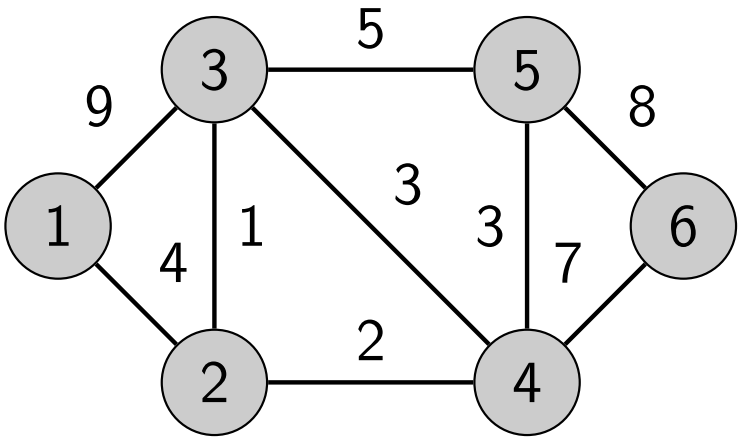


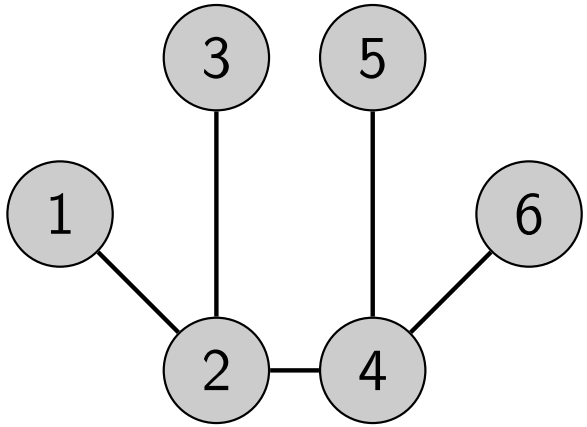

3
5
5
9
8
3
1
3
6
4
1
7
2
2
4

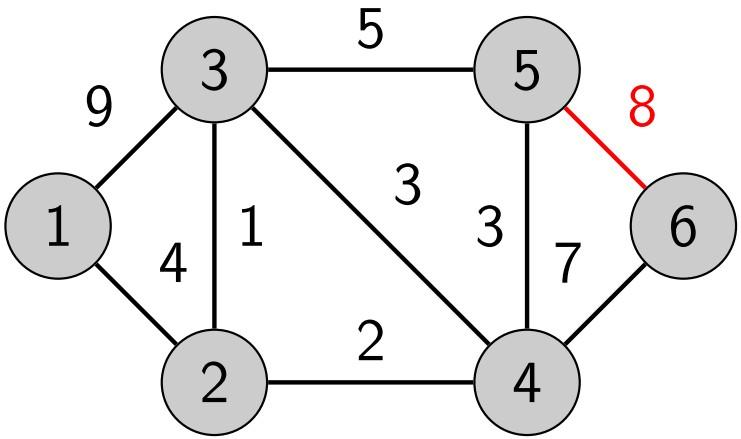

3
5
1
6
2
4

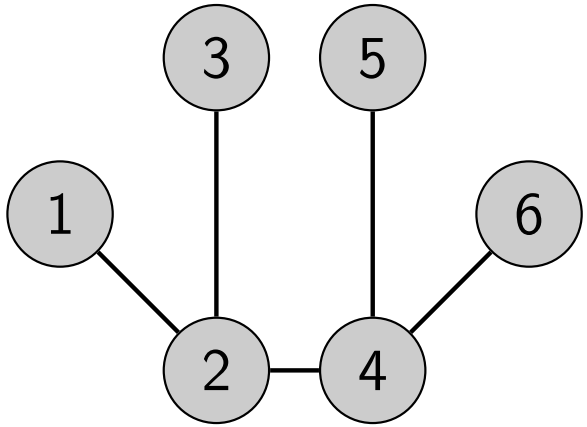
Algorytm Kruskala

Przykład działania



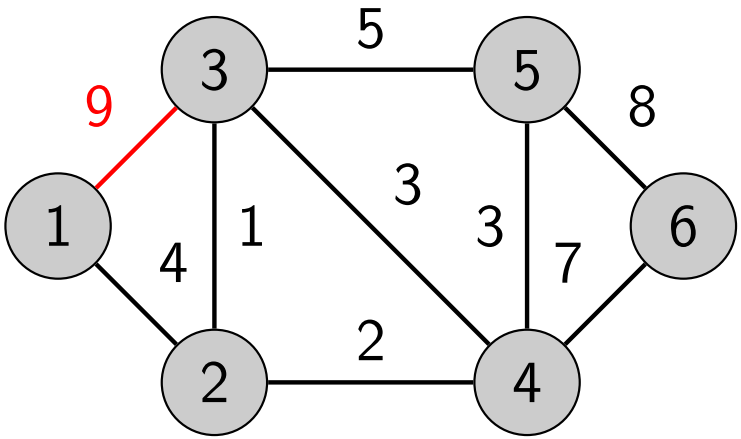


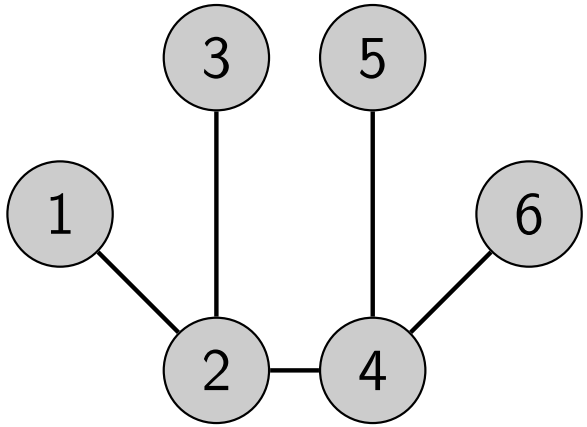

1
2
3
4
5
6
9
5
8
3
4
1
3
7
2


1
2
3
4
5
6

Algorytm Kruskala

Przykład działania





Niech G będzie spójnym ważonym grafem, a T będzie podgrafem G wyprodukowanym, przez algorytm.

Niech G będzie spójnym ważonym grafem, a T będzie podgrafem G wyprodukowanym, przez algorytm. T nie może zawierać cyklu, gdyż z definicji wynika, że krawędzie tworzące cykl nie są dodawane. T musi być spójne, gdyż każda napotykana krawędź łącząca dwie składowe T zostaje do niego dodana. Zatem T jest drzewem rozpinającym grafu G.

Niech G będzie spójnym ważonym grafem, a T będzie podgrafem G wyprodukowanym, przez algorytm. T nie może zawierać cyklu, gdyż z definicji wynika, że krawędzie tworzące cykl nie są dodawane. T musi być spójne, gdyż każda napotykana krawędź łącząca dwie składowe T zostaje do niego dodana. Zatem T jest drzewem rozpinającym grafu G.
Dowód minimalności T robi się indukcyjnie pokazując, że w każdym momencie algorytmu zbiór F krawędzi T zawiera się w zbiorze krawędzi pewnego z minimalnych drzew rozpinających grafu G.
Baza indukcyjna jest trywialna ($F = \varnothing$).

Niech G będzie spójnym ważonym grafem, a T będzie podgrafem G wyprodukowanym, przez algorytm. T nie może zawierać cyklu, gdyż z definicji wynika, że krawędzie tworzące cykl nie są dodawane. T musi być spójne, gdyż każda napotykana krawędź łącząca dwie składowe T zostaje do niego dodana. Zatem T jest drzewem rozpinającym grafu G.
Dowód minimalności T robi się indukcyjnie pokazując, że w każdym momencie algorytmu zbiór F krawędzi T zawiera się w zbiorze krawędzi pewnego z minimalnych drzew rozpinających grafu G.
Baza indukcyjna jest trywialna ($F = \varnothing$). Krok pozostawiam jako ćwiczenie.

Algorytm Kruskala

```
struct krawedz {
  int v, u; long long w;
  bool operator<(krawedz e) { return w < e.w; }
};
```

Algorytm Kruskala

```
struct krawedz {
  int v, u; long long w;
  bool operator<(krawedz e) { return w < e.w; }
};
...
sort(krawedzie.begin(), krawedzie.end());
for (auto e : krawedzie) {
  if (polaczone(e.v, e.u)) continue;
  polacz(e.v, e.u);
  wynik.emplace_back(e);
}
```

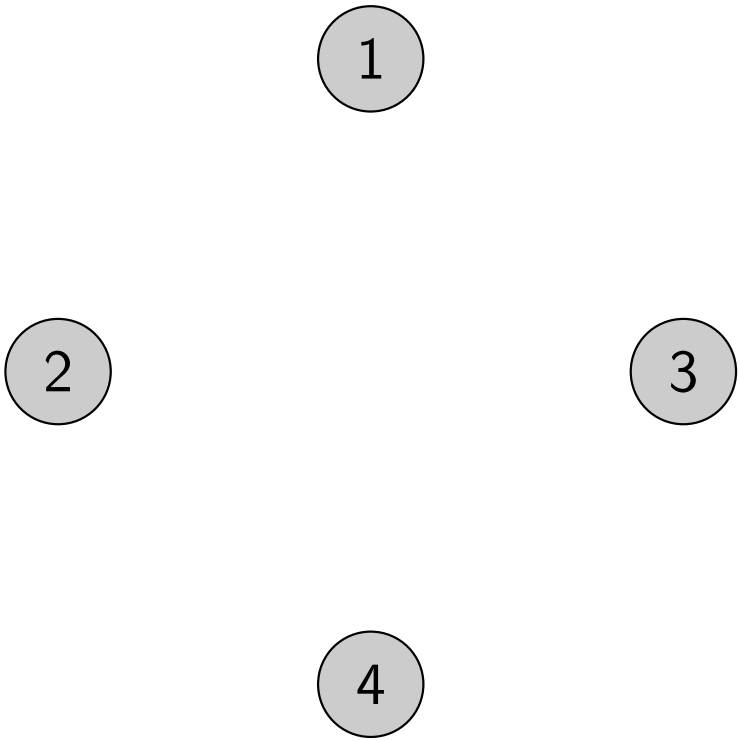
Struktura zbiorów rozłącznych (DSU)

- Wprowadzenie
- Naiwne rozwiązanie
- Łączenie według rangi/rozmiaru
- Kompresja ścieżek

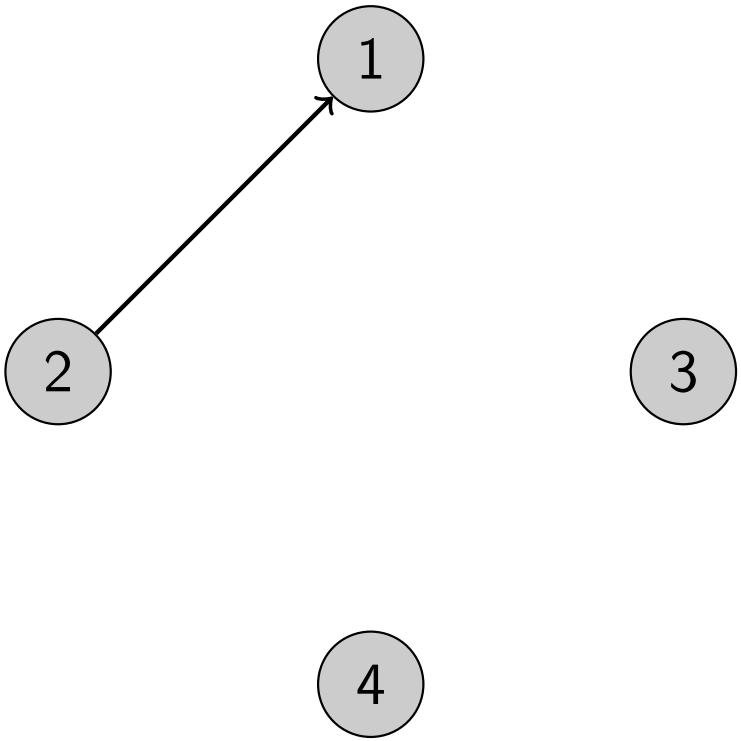
Struktura zbiorów rozłącznych (ang. *Disjoint Set Union* lub *Union Find*) pozwalająca na 3 typy operacji:

- `make(v)` – tworzy nowy zbiór zawierający element v;
- `union(v, u)` – scala zbiory zawierające v i u w jeden;
- `find(v)` – zwraca reprezentanta (zwanego też liderem) zbioru, do którego należy v.

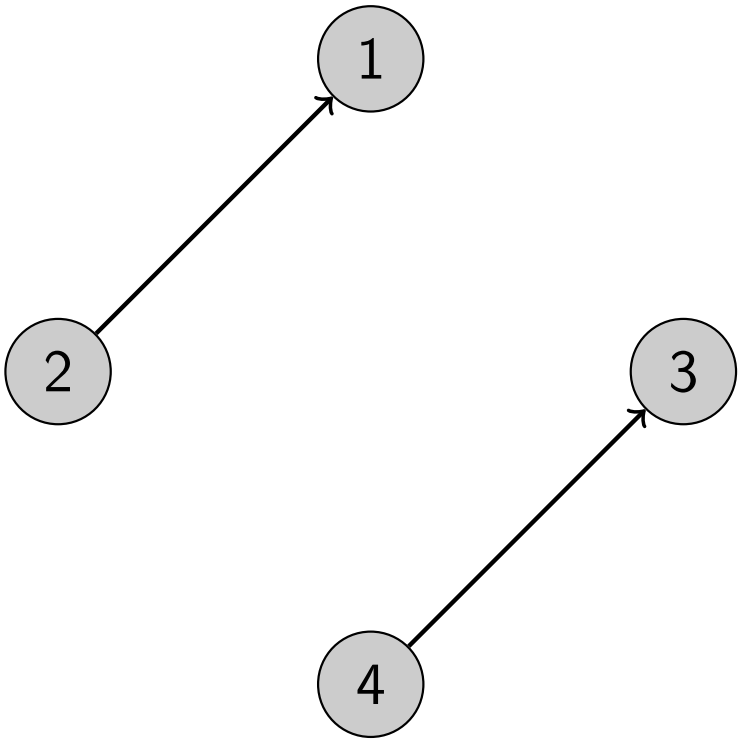
Zbiory będziemy przechowywać w postaci drzew: każdemu drzewo będzie odpowiadało jednemu zbiorowi. Korzeń drzewa będzie reprezentantem zbioru.



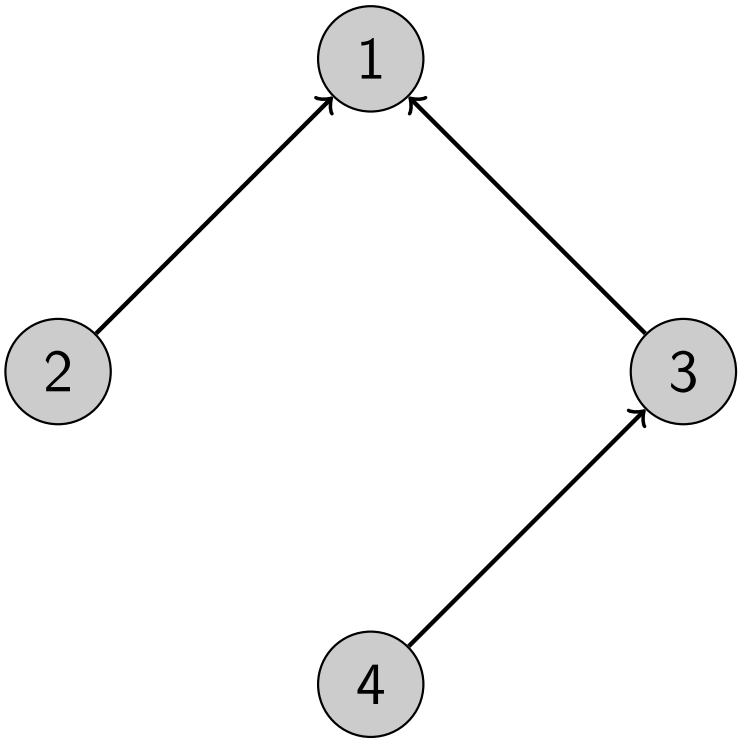
Zbiory będziemy przechowywać w postaci drzew: każdemu drzewo będzie odpowiadało jednemu zbiorowi. Korzeń drzewa będzie reprezentantem zbioru.



Zbiory będziemy przechowywać w postaci drzew: każdemu drzewo będzie odpowiadało jednemu zbiorowi. Korzeń drzewa będzie reprezentantem zbioru.



Zbiory będziemy przechowywać w postaci drzew: każdemu drzewo będzie odpowiadało jednemu zbiorowi. Korzeń drzewa będzie reprezentantem zbioru.



Naiwne rozwiązanie

Bardzo łatwo można przechowywać ojca każdego wierzchołka (ojcem reprezentanta będzie on sam).

Naiwne rozwiązanie

Bardzo łatwo można przechowywać ojca każdego wierzchołka (ojcem reprezentanta będzie on sam).

```
int rep[MAX_N];
int Find(int v) {
  return rep[v] == v ? v : Find(rep[v]);
}
void Union(int v, int u) {
  rep[Find(v)] = Find(u);
}
```

Naiwne rozwiązanie

Bardzo łatwo można przechowywać ojca każdego wierzchołka (ojcem reprezentanta będzie on sam).

```
int rep[MAX_N];
int Find(int v) {
  return rep[v] == v ? v : Find(rep[v]);
}
void Union(int v, int u) {
  rep[Find(v)] = Find(u);
}
```

To rozwiązanie niestety może być czasem bardzo wolne. Nietrudno wymyślić przykład, w którym drzewa będą formować długie łańcuchy. W pesymistycznym przypadku dla jednego zapytania mamy złożoność $O(n)$.

Łączenie według rangi/rozmiaru

W naiwnym rozwiązaniu zawsze dołączaliśmy lewy zbiór do prawego. Algorytm można znacząco przyspieszyć, dołączając mniejszy ze zbiorów do większego lub tego z mniejszą rangą do tego z większą (ranga zbioru odpowiada wysokości drzewa).

Łączenie według rangi/rozmiaru

W naiwnym rozwiązaniu zawsze dołączaliśmy lewy zbiór do prawego. Algorytm można znacząco przyspieszyć, dołączając mniejszy ze zbiorów do większego lub tego z mniejszą rangą do tego z większą (ranga zbioru odpowiada wysokości drzewa).

```
struct { int rep, ranga; } f[MAX_N];
int Find(int v) {
  return f[v].rep == v ? v : Find(f[v].rep);
}
```

Łączenie według rangi/rozmiaru

W naiwnym rozwiązaniu zawsze dołączaliśmy lewy zbiór do prawego. Algorytm można znacząco przyspieszyć, dołączając mniejszy ze zbiorów do większego lub tego z mniejszą rangą do tego z większą (ranga zbioru odpowiada wysokości drzewa).

```
struct { int rep, ranga; } f[MAX_N];
int Find(int v) {
  return f[v].rep == v ? v : Find(f[v].rep);
}
void Union(int v, int u) {
  v = Find(v), u = Find(u);
  if (f[v].ranga > f[u].ranga) swap(v, u);
  f.rep[v] = u;
  f[u].ranga = max(f[v].ranga + 1, f[u].ranga);
}
```

Łączenie według rangi/rozmiaru

W naiwnym rozwiązaniu zawsze dołączaliśmy lewy zbiór do prawego. Algorytm można znacząco przyspieszyć, dołączając mniejszy ze zbiorów do większego lub tego z mniejszą rangą do tego z większą (ranga zbioru odpowiada wysokości drzewa).

```
struct { int rep, ranga; } f[MAX_N];
int Find(int v) {
  return f[v].rep == v ? v : Find(f[v].rep);
}
void Union(int v, int u) {
  v = Find(v), u = Find(u);
  if (f[v].ranga > f[u].ranga) swap(v, u);
  f.rep[v] = u;
  f[u].ranga = max(f[v].ranga + 1, f[u].ranga);
}
```

Tak proste usprawnienie zbija złożoność do $O(\log n)$.

Złożoność funkcji `Union` jest w oczywisty sposób taka sama jak funkcji `Find`. Ta z kolei zależy jedynie od rangi zbioru. Naszym celem będzie udowodnienie, że w tak powstałych zbiorach $r(A) \leq \log |A|$.

Złożoność funkcji `Union` jest w oczywisty sposób taka sama jak funkcji `Find`. Ta z kolei zależy jedynie od rangi zbioru. Naszym celem będzie udowodnienie, że w tak powstałych zbiorach $r(A) \leq \log |A|$.

Dowód przez indukcje po rozmiarze zbioru:

$$|A| = 1 \implies r(A) = 0 \leq \log 1 = \log |A|$$

Złożoność funkcji `Union` jest w oczywisty sposób taka sama jak funkcji `Find`. Ta z kolei zależy jedynie od rangi zbioru. Naszym celem będzie udowodnienie, że w tak powstałych zbiorach $r(A) \leq \log |A|$.

Dowód przez indukcje po rozmiarze zbioru:

$$|A| = 1 \implies r(A) = 0 \leq \log 1 = \log |A|$$

Bez straty ogólności niech $|A| \leq |B|$, niech T będzie zbiorem otrzymanym poprzez wykonanie `Union` na zbiorach A i B.

$$r(T) = \max(r(B), 1 + r(A)) \leq \max(\log |B|, 1 + \log |A|)$$
$$\log |B| \leq \log (|A| + |B|) \implies r(T) \leq \log (|A| + |B|) = \log |T|$$
$$\log_2 2 + \log |A| = \log (|A| \cdot 2) \leq \log (|A| + |B|) = \log |T|$$

Kompresja ścieżek

Rozwiązanie można jeszcze bardziej usprawnić osiągając średnią złożoność rzędu $O(\alpha(n))$, gdzie $\alpha(n)$ to odwrotna funkcja Ackermanna (dla $n \leq 10^{600}, \alpha(n) \leq 4$). Dowód złożoności jest bardzo długi, więc go tutaj nie pokażę.

Kompresja ścieżek

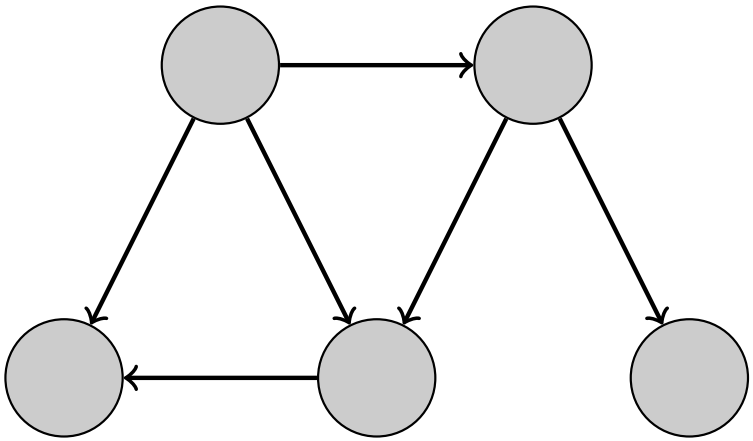
Rozwiązanie można jeszcze bardziej usprawnić osiągając średnią złożoność rzędu $O(\alpha(n))$, gdzie $\alpha(n)$ to odwrotna funkcja Ackermanna (dla $n \leq 10^{600}, \alpha(n) \leq 4$). Dowód złożoności jest bardzo długi, więc go tutaj nie pokażę.
Usprawnienie wygląda tak:

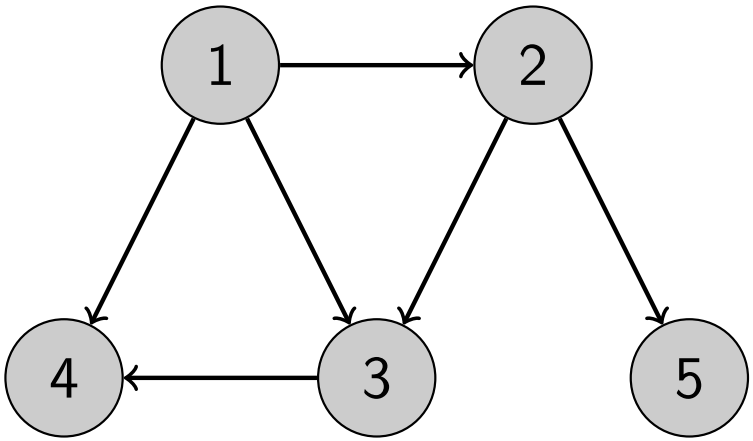
```
int Find(int v) {
  return f[v].rep == v ? v
                       : f[v].rep = Find(f[v].rep);
}
```

Sortowanie topologiczne (Toposort)

- Wprowadzenie
- Algorytm

Dany jest graf skierowany $G = \langle V, E \rangle$. Należy znaleźć taki porządek wierzchołków, aby każda krawędź prowadziła z wierzchołka o mniejszym indeksie do wierzchołka z większym. Innymi słowy, chcemy znaleźć taką permutację wierzchołków (porządek topologiczny), który odpowiada porządkowi zdefiniowanemu przez krawędzie (jeśli istnieje krawędź z wierzchołka v do wierzchołka u, to v powinno być przed u).




1
2
4
3
5

Warto zauważyć, że może istnieć wiele możliwych porządków topologicznych, lub może też nie istnieć żaden. Wiele poprawnych porządków topologicznych widać choćby na pokazanym przykładzie, natomiast porządek topologiczny nie istnieje, gdy w grafie istnieje cykl.

Algorytm

Aby rozwiązać problem, najłatwiej jest użyć przeszukiwania w głąb. Przypuśćmy, że dany graf jest acykliczny. Będziemy tworzyli listę, na której początek będziemy dodawali wierzchołki, z których wychodzi DFS.

Algorytm

Aby rozwiązać problem, najłatwiej jest użyć przeszukiwania w głąb. Przypuśćmy, że dany graf jest acykliczny. Będziemy tworzyli listę, na której początek będziemy dodawali wierzchołki, z których wychodzi DFS.

```
vector<int> toposort; // odwrotna kolejnosc
void dfs(int v) {
  visited[v] = true;
  for (auto u : G[v]) {
    if (!visited[u])
      dfs(v);
  }
  toposort.emplace_back(v);
}
...
reverse(toposort.begin(), toposort.end());
```

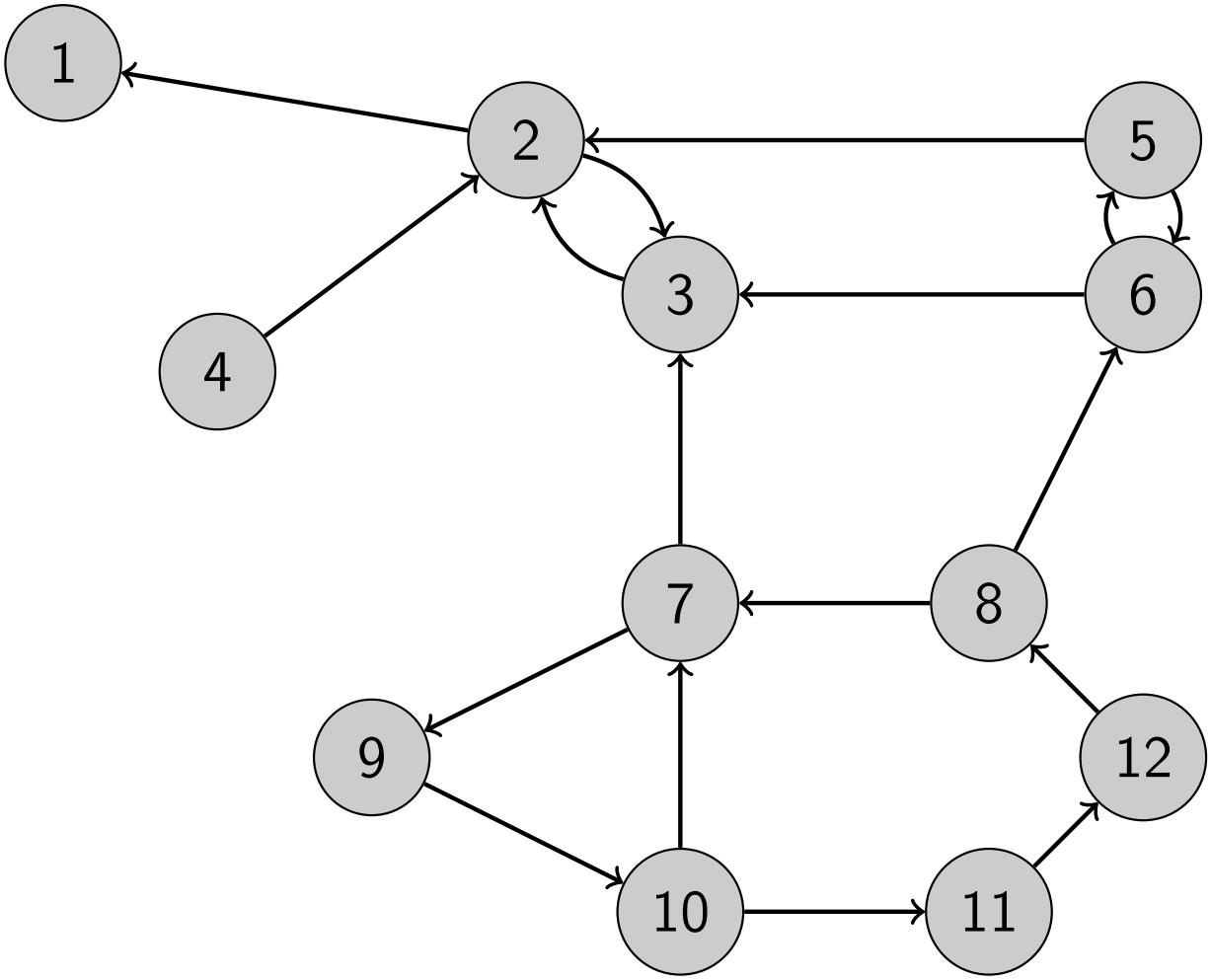
Silnie spójne składowe (SCC)

- Wprowadzenie
- Algorytm Kosaraju
- Graf silnie spójnych składowych (Condensation graph)

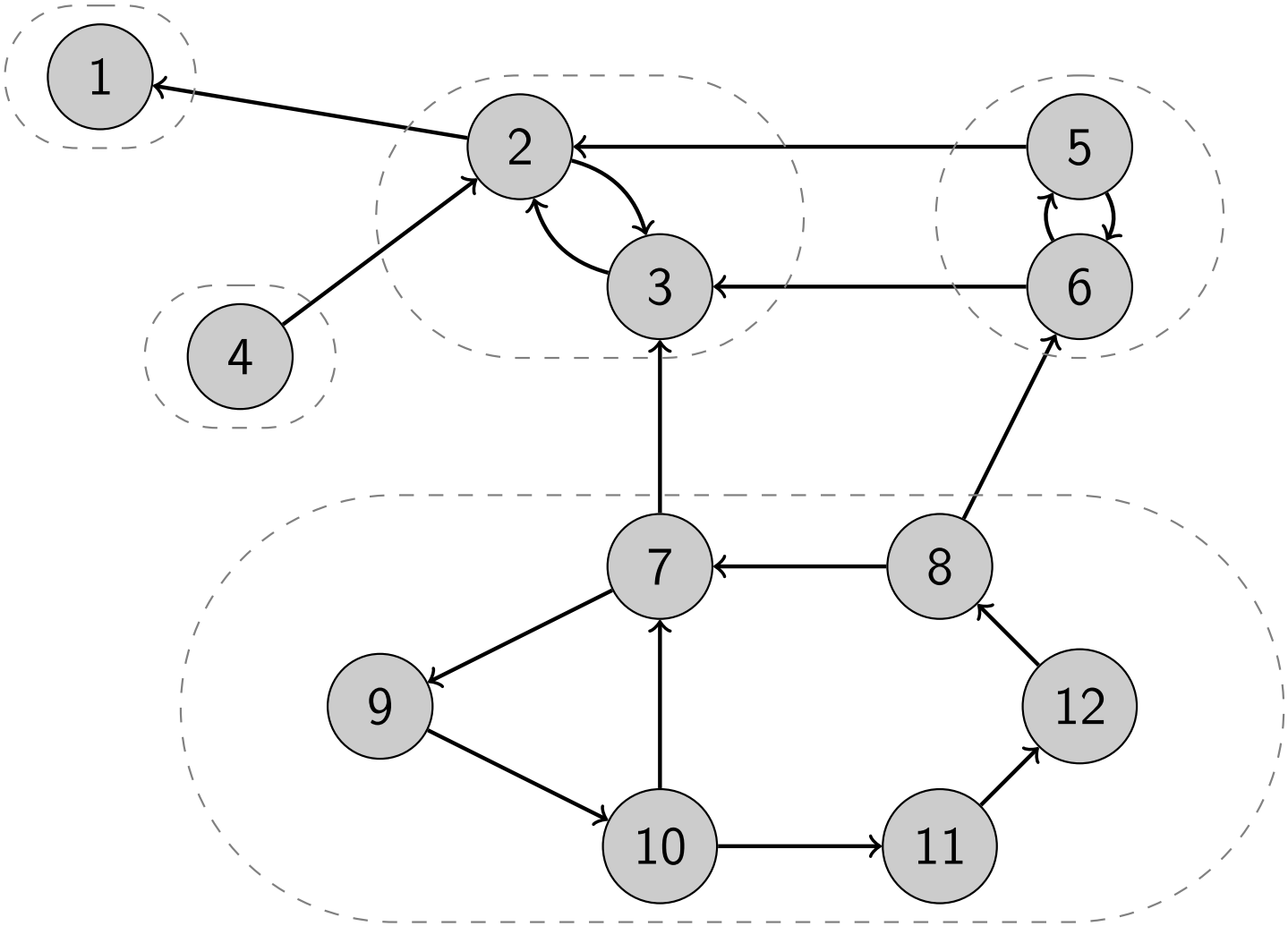
Dany jest skierowany graf $G = \langle V, E \rangle$. Mogą istnieć w nim pętle i krawędzie wielokrotne. Silnie spójna składowa to maksymalny podzbiór wierzchołków C, taki że między każdą parą jego wierzchołków istnieje ścieżka $(\forall_{v,u \in C} v \rightsquigarrow u \wedge u \rightsquigarrow v)$.

Dany jest skierowany graf $G = \langle V, E \rangle$. Mogą istnieć w nim pętle i krawędzie wielokrotne. Silnie spójna składowa to maksymalny podzbiór wierzchołków C, taki że między każdą parą jego wierzchołków istnieje ścieżka $(\forall_{v,u \in C} v \rightsquigarrow u \wedge u \rightsquigarrow v)$.
Jest oczywiste, że dwie silnie spójne składowe grafu G są rozłączne, inaczej nie byłyby to maksymalne podzbiory. Możemy w takim razie stworzyć nowy graf G^{SCC}, w którym wierzchołki to będą silnie spójne składowe G. Graf silnie spójnych składowych jest acykliczny, ponieważ jeśli istniałby cykl, to nie byłyby to maksymalne podgrafy.

Przykład



Przykład



Niech $SCC[G] = \{S_1, S_2, ..., S_s\}$ będzie rodziną zbiorów wierzchołków grafu G takich, że wierzchołki z jednego zbioru tworzą silnie spójną składową. Niech $\psi(S_i) = \max\{tout(v) | v \in S_i\}$. Niech $A, B \in SCC[G]$. Wtedy

$$\forall_{A,B \in SCC[G], A \neq B} \psi(A) < \psi(B) \implies \neg \exists A \rightsquigarrow B.$$

Niech G^T będzie grafem transponowanym G, czyli grafem z odwróconymi zwrotami krawędzi. Wtedy

$$SCC[G] = SCC[G^T].$$

Dowody obu obserwacji są dość łatwe i pozostawiam je jako ćwiczenie.

Algorytm Kosaraju

Algorytm zaproponowany przez Kosaraju opiera się w głównej mierze na powyższych obserwacjach i wygląda tak:

1. wyznaczamy czasy wyjścia wierzchołków G;
2. wywołujemy DFS dla G^T w kolejności malejących czasów wyjścia. Wszystkie wierzchołki w jednym drzewie przeszukiwania w głąb należą do jednej silnie spójnej składowej.

```
vector<int> G[MAX_N], Gt[MAX_N];
bool visited[MAX_N];
vector<int> order;
void dfs1(int v) {
  visited[v] = true;
  for (auto u : G[v])
    if (!visited[u])
      dfs1(u);
  order.emplace_back(v);
}
...
for (int v = 1; v <= n; v++)
  if (!visited[v])
    dfs1(v);
```

Algorytm Kosaraju

```
int scc[MAX_N], counter = 0;
void dfs2(int v) {
  scc[v] = counter;
  for (auto u : Gt[v])
    if (scc[u] == 0)
      dfs2(u);
}
...
reverse(order.begin(), order.end());
for (auto v : order) {
  if (!scc[v]) {
    counter++;
    dfs2(v);
  }
}
```

Graf silnie spójnych składowych (Condensation graph)

```
vector<int> Gscc[counter + 7];
for (int v = 1; v <= n; v++) {
  for (auto u : G[v])
    Gscc[scc[v]].emplace_back(scc[u]);
}
```

Graf silnie spójnych składowych (Condensation graph)

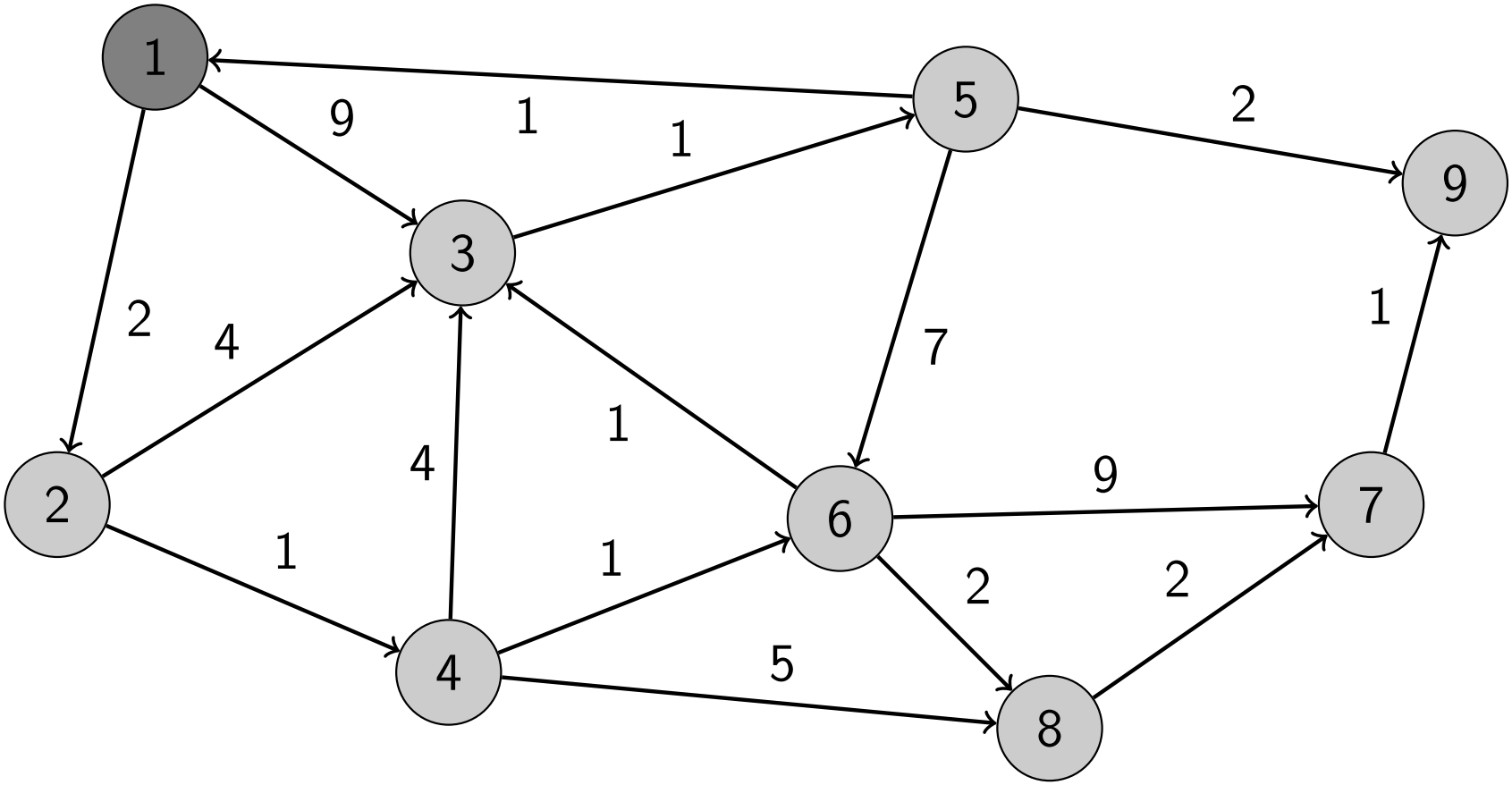
```
vector<int> Gscc[counter + 7];
for (int v = 1; v <= n; v++) {
  for (auto u : G[v])
    Gscc[scc[v]].emplace_back(scc[u]);
}
```

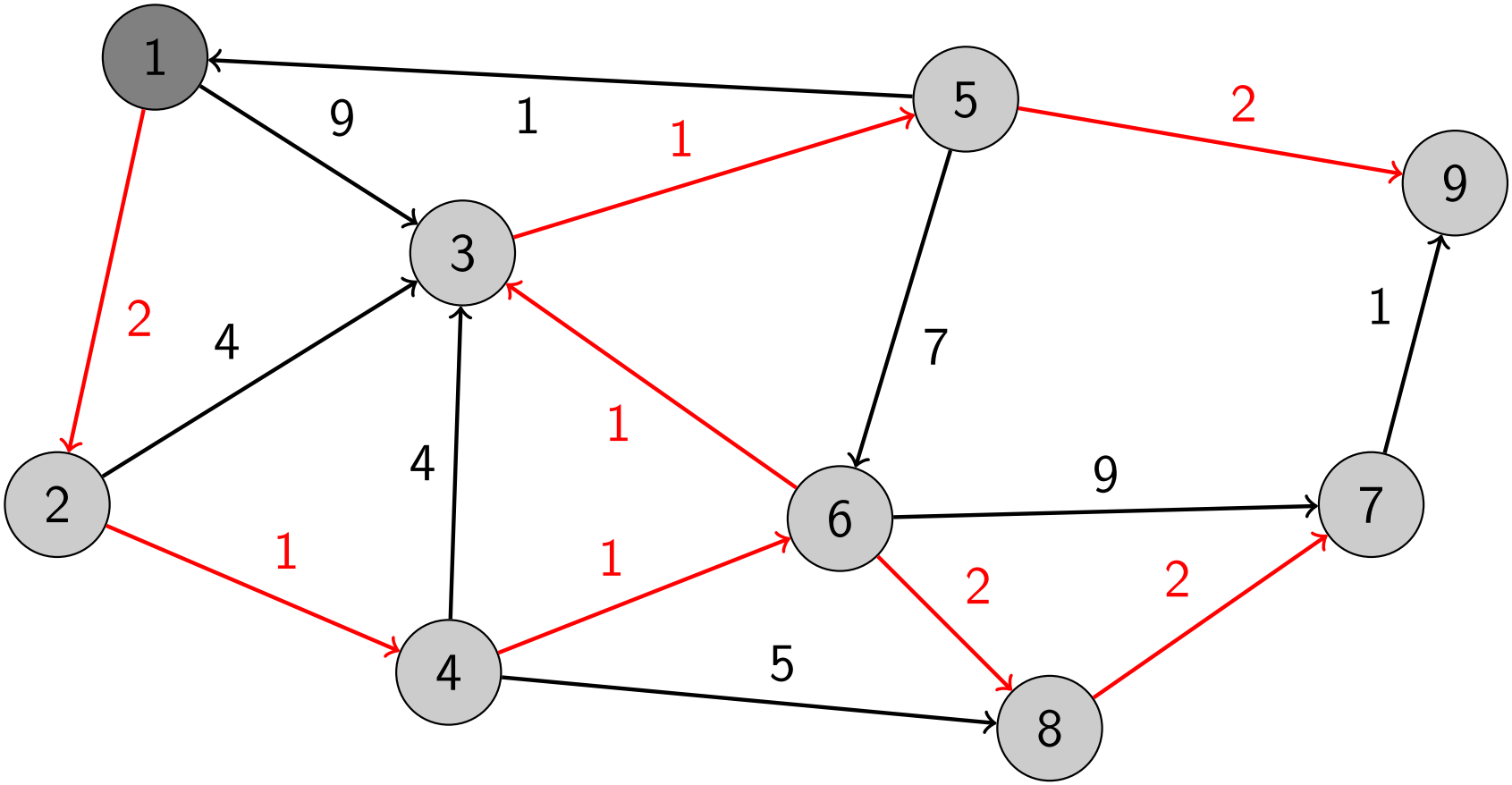
Warto zauważyć, że przy takiej implementacji możliwe jest istnienie bardzo wielu wtórnych krawędzi. To jednak z reguły nie jest problemem (przynajmniej w zadaniach konkursowych).

Najlżejsze (najkrótsze) ścieżki

- Wprowadzenie
- Algorytm Dijkstry

Dany jest spójny, ważony graf $G = \langle V, E \rangle$ i wyróżniony wierzchołek $s \in V$. Wszystkie wagi krawędzi są nieujemne. Naszym zadaniem jest znalezienie dla każdego wierzchołka $v \in V$ najlżejszej ścieżki $s \rightsquigarrow v$, gdzie wagę ścieżki definiujemy jako sumę wag jej krawędzi.


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


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

Algorytm Dijkstry

Zacznijmy od kilku definicji:

- $w^*(v)$ = waga najlżejszej ścieżki $s \rightsquigarrow v$;
- $w'(v)$ = waga najlżejszej dotychczas znalezionej ścieżki $s \rightsquigarrow v$;
- $L = \{v \in V \big| w^*(v) = w'(v)\}$, $R = V \setminus L$.

Algorytm Dijkstry

Zacznijmy od kilku definicji:

- $w^*(v) =$ waga najlżejszej ścieżki $s \rightsquigarrow v$;
- $w'(v) =$ waga najlżejszej dotychczas znalezionej ścieżki $s \rightsquigarrow v$;
- $L = \{v \in V \big| w^*(v) = w'(v)\}$, $R = V \setminus L$.

Zachodzi następujący niezmiennik:

- $V = L \cup R$;
- $s \in L$;
- $\forall_{u \in L, v \in R} w'(u) = w^*(u) \leq w^*(v)$;
- $\forall_{v \in R} w'(v) = \min\{w^*(u) + w(\{u, v\}) \big| u \in L, \{u, v\} \in E\} \cup \{\infty\}$;

Algorytm Dijkstry

Algorytm, który Edsger W. Dijkstra zaproponował w 1959 roku wygląda w następujący sposób:

Algorytm Dijkstry

Algorytm, który Edsger W. Dijkstra zaproponował w 1959 roku wygląda w następujący sposób:
Dopóki $R \neq \varnothing$:

Algorytm Dijkstry

Algorytm, który Edsger W. Dijkstra zaproponował w 1959 roku wygląda w następujący sposób:
Dopóki $R \neq \varnothing$:

- wybieramy wierzchołek $v \in R$ o najmniejszym w';

Algorytm Dijkstry

Algorytm, który Edsger W. Dijkstra zaproponował w 1959 roku wygląda w następujący sposób:
Dopóki $R \neq \varnothing$:

- wybieramy wierzchołek $v \in R$ o najmniejszym w';
- odejmujemy go od zbioru R i dodajemy do zbioru L;

Algorytm Dijkstry

Algorytm, który Edsger W. Dijkstra zaproponował w 1959 roku wygląda w następujący sposób:
Dopóki $R \neq \varnothing$:

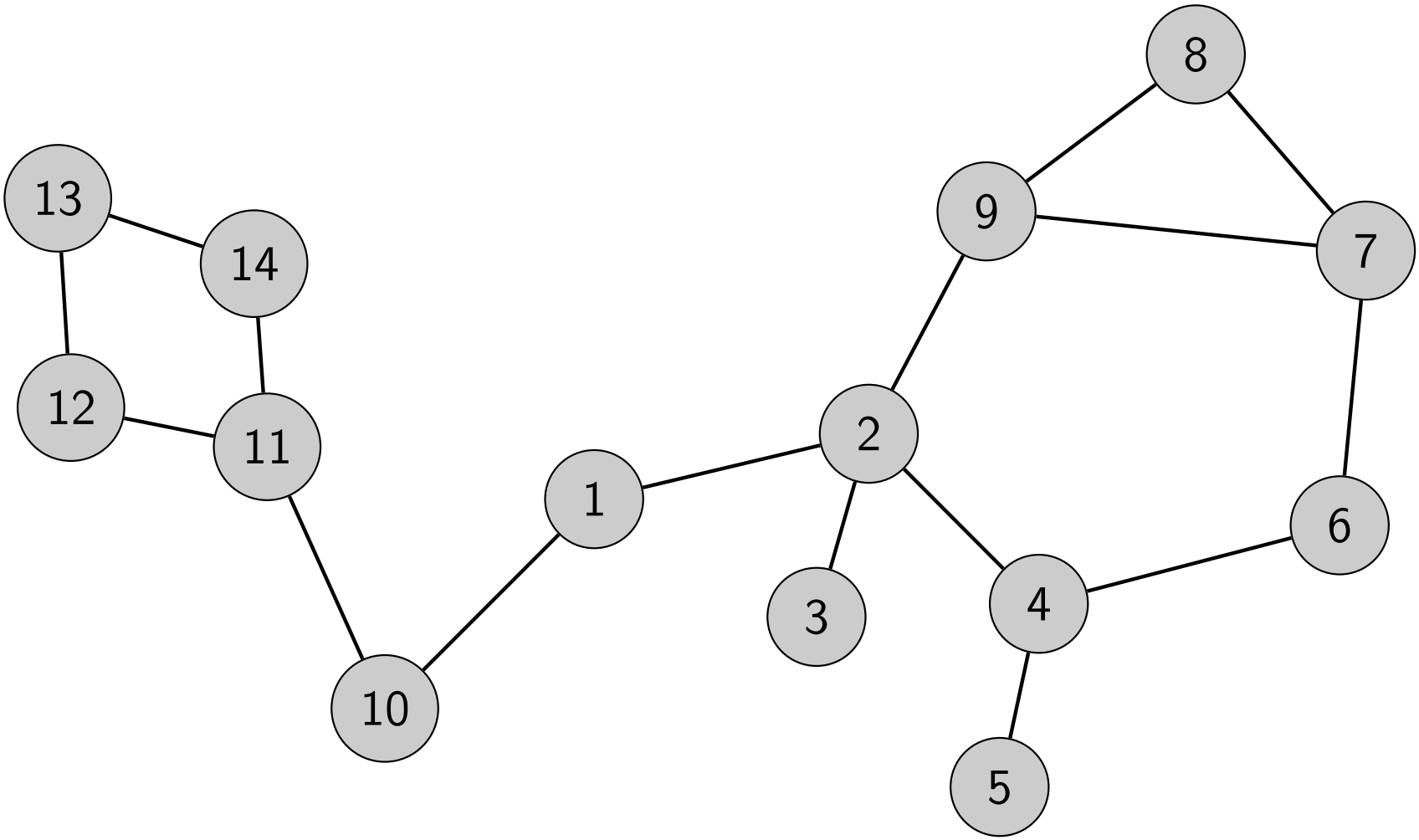
- wybieramy wierzchołek $v \in R$ o najmniejszym w';
- odejmujemy go od zbioru R i dodajemy do zbioru L;
- dla każdego jego sąsiada $u \in R$ ustawiamy $w'(u) = \min(w'(u), w^*(v) + w(\{v, u\}))$.

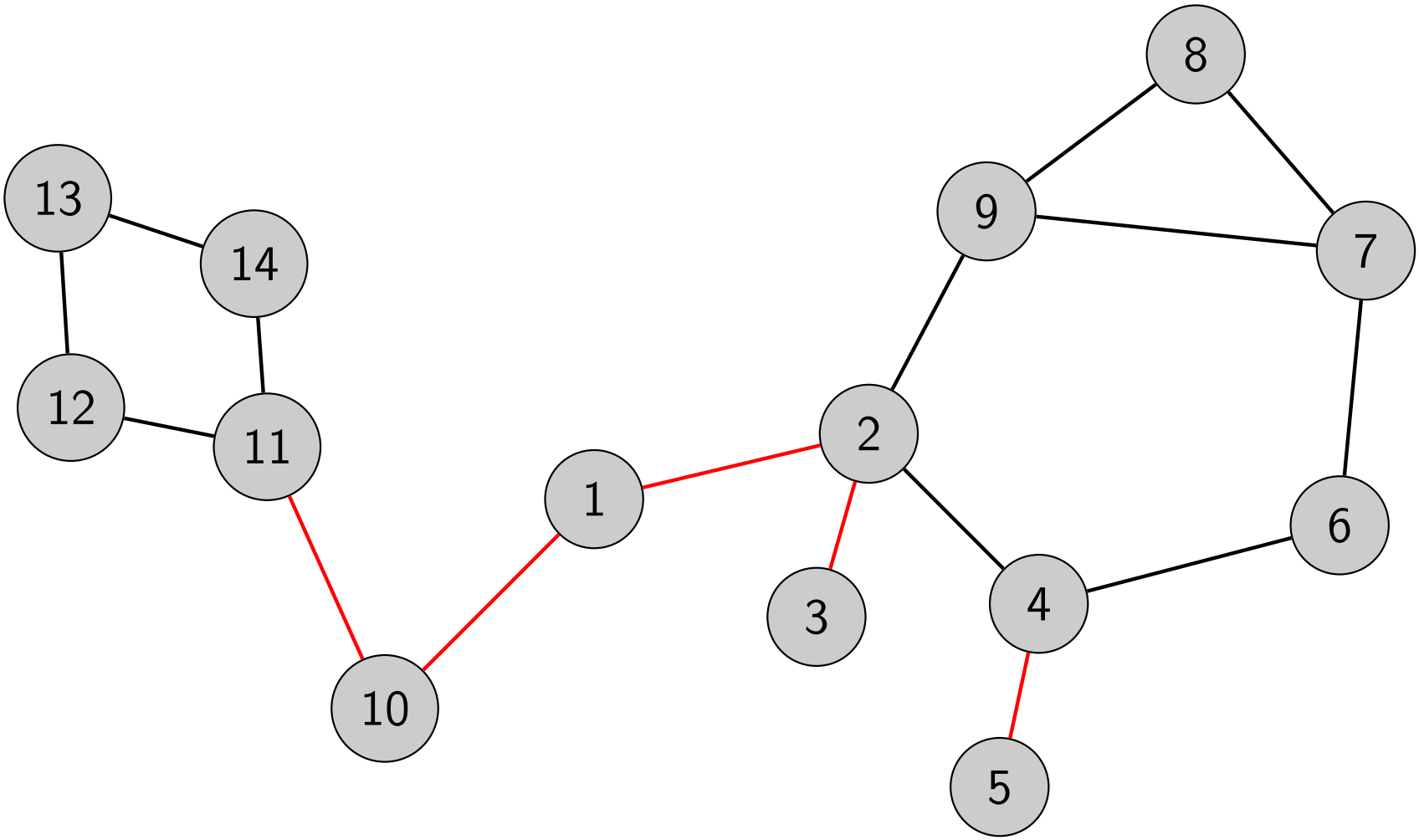
```
struct myPair {
  int v, w;
  bool operator>(myPair &p) { return w < p.w; }
};
...
priority_queue<myPair> R({{s, 0}});
while (!R.empty()) {
  myPair v = R.top(); R.pop();
  for (int u : G[v]) {
    if (v.w + u.second < w[u.first]) {
      w[u.first] = v.w + u.second;
      R.push({u.first, w[u.first]});
    }
  }
}
```

Parametr low (mosty i punkty artykulacji)

- Wprowadzenie
- Algorytm

Dany jest nieskierowany graf $G = \langle V, E \rangle$. Naszym zadaniem jest wyznaczenie wszystkich krawędzi, których usunięcie zwiększa liczbę spójnych składowych G.


8
13
9
7
14
12
11
2
1
6
3
4
10
5


8
13
9
7
14
12
11
2
1
6
3
4
10
5

Algorytm

Wybierzmy dowolny wierzchołek *root* i odpalmy na nim *DFS*-a.
Łatwo jest udowodnić następujący fakt:

Algorytm

Wybierzmy dowolny wierzchołek *root* i odpalmy na nim *DFS*-a. Łatwo jest udowodnić następujący fakt:

- Powiedzmy, że jesteśmy w wierzchołku v. Krawędź (v, u), gdzie u jest synem v w drzewie DFS, jest mostem wtedy i tylko wtedy, gdy żaden z wierzchołków poddrzewa DFS wierzchołka u nie ma krawędzi wstecznej do v lub jego przodka w drzewie DFS.

Algorytm

Oznaczmy czas wejścia (preorder) wierzchołka v poprzez $tin[v]$.

Oznaczmy czas wejścia (preorder) wierzchołka v poprzez $tin[v]$.

$$low[v] = \min \begin{cases} tin[v] & \\ tin[p] & \forall p \text{ takich, że krawędź } (v, p) \text{ jest wsteczna} \\ low[u] & \forall u \text{ takich, że krawędź } (v, u) \text{ jest w drzewie} \end{cases}$$

Oznaczmy czas wejścia (preorder) wierzchołka v poprzez $tin[v]$.

$$low[v] = \min \begin{cases} tin[v] & \\ tin[p] & \forall p \text{ takich, że krawędź } (v, p) \text{ jest wsteczna} \\ low[u] & \forall u \text{ takich, że krawędź } (v, u) \text{ jest w drzewie} \end{cases}$$

Przy tak zdefiniowanej funkcji *low* krawędź (v, u) w drzewie DFS jest mostem wtedy i tylko wtedy, gdy $tin[v] < low[u]$.
Złożoność algorytmu jest oczywiście taka jak złożoność *DFS*-a, czyli $O(n + m)$.

```
vector<int> G[MAX_N];
bool visited[MAX_N];
int low[MAX_N], tin[MAX_N], timer = 0;
void dfs(int v, int p = -1) {
  visited[v] = true;
  tin[v] = low[v] = ++timer;
  for (auto u : G[v]) {
    if (u == p) continue;
    if (visited[u]) {
      low[v] = min(low[v], tin[u]);
    } else {
      dfs(u, v);
      low[v] = min(low[v], low[u]);
      if (tin[v] < low[u]) JEST_MOSTEM(v, u);
    }
  }
}
```

Drzewa przedziałowe

- Drzewa przedział-punkt
- Drzewa przedział-przedział

Dany jest ciąg c_n i łączna, dwuargumentowa operacja $\diamond$. Łączna to taka, że $a \diamond (b \diamond c) = (a \diamond b) \diamond c$. Mamy wykonać na tym ciągu operacje postaci:

- zmień c_i w x;
- podaj wynik na przedziale $[l, r]$, czyli $c_l \diamond c_{l+1} \diamond \ldots \diamond c_r$.

Drzewa przedziałowe są strukturą pozwalającą na wykonywanie tych operacji w czasie $O(\log n)$.

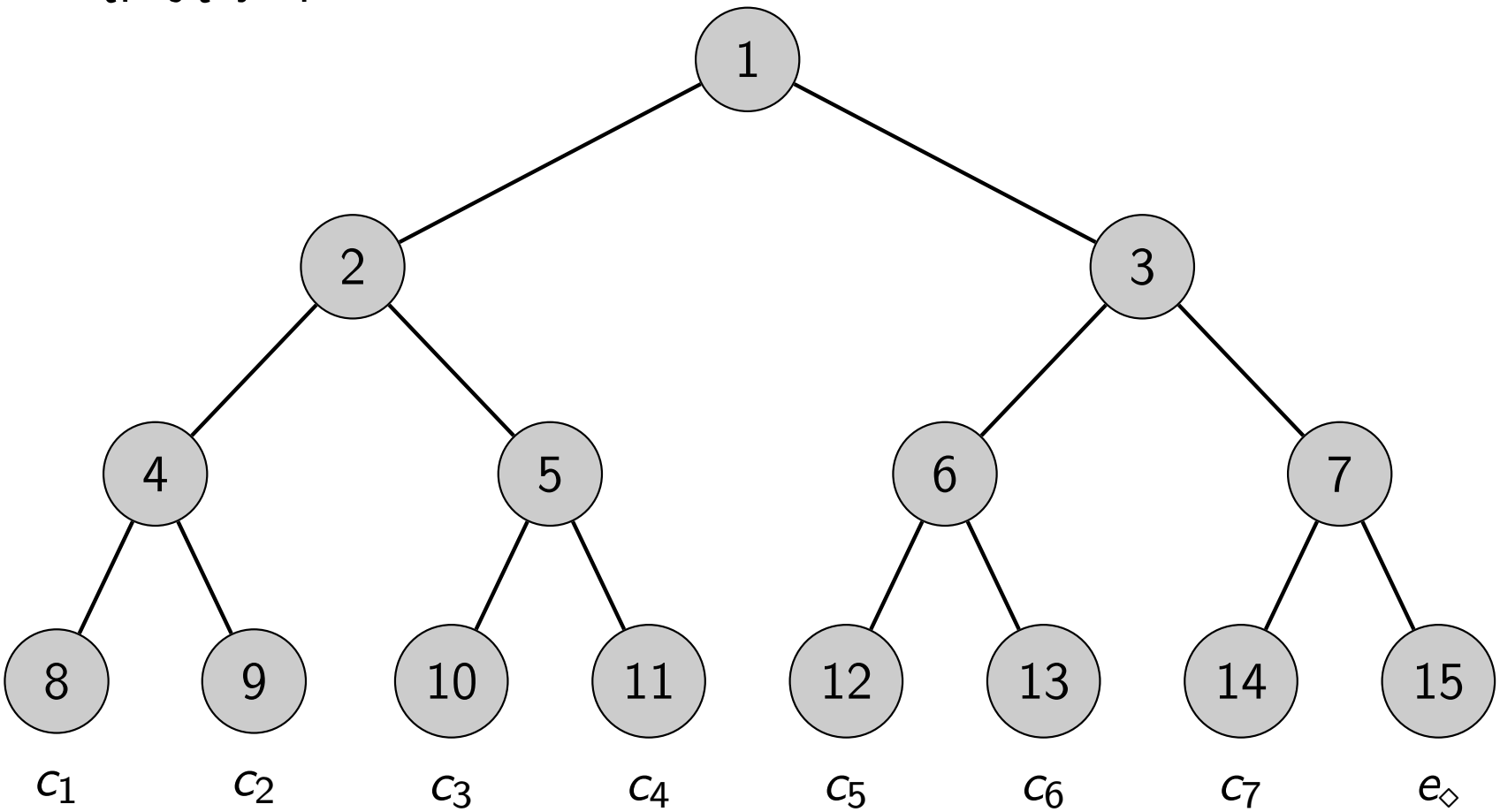
Zbudujemy na ciągu c_n pełne drzewo binarne (jeśli trzeba, to dopiszemy elementy neutralne operacji $\diamond$, aby było 2^k liści). Drzewo będzie przechowywane w tablicy *tree*, zdefiniowanej w następujący sposób:

$$tree[v] = \begin{cases} c_{v-leafs} & \text{gdy } v \geq leafs; \\ tree[2v] \diamond tree[2v+1] & \text{w przeciwnym przypadku.} \end{cases}$$

Wiemy wówczas, że $tree[v]$ trzyma wynik operacji $\diamond$ wykonanej na wszystkich liściach poddrzewa wierzchołka v, a w szczególności $tree[1] = c_1 \diamond c_2 \diamond \ldots \diamond c_n$, gdyż liście które sztucznie dodamy zawierają elementy neutralne $e_\diamond$, a oczywiście $x \diamond e_\diamond = x$ dla każdego x z dziedziny.

Dla ciągu 7-elementowego drzewo mogło by wyglądać w następujący sposób.

Dla ciągu 7-elementowego drzewo mogło by wyglądać w następujący sposób.



Aktualizacja

Jeśli chcemy zmienić c_i, to zmieniamy odpowiadający mu liść. Po aktualizacji wartości wierzchołka v należy zaktualizować wartość jego ojca – wierzchołek $\lfloor \frac{v}{2} \rfloor$.

Zapytanie

Aby odpowiedzieć na zapytanie postaci $[l, r]$ należy wyznaczyć wierzchołki bazowe przedziału – wierzchołki, których przedział zawiera się w $[l, r]$, a przedział ich ojca nie. Wynikiem jest $tree[b_1] \diamond ... \diamond tree[b_k]$, gdzie ciąg b_k, to wyznaczony ciąg wierzchołków bazowych.

```
void update(int v, T x) {
  tree[(v += M - 1)] = x;
  while ((v /= 2))
    tree[v] = tree[v * 2] ◇ tree[v * 2 + 1];
}
```

```
T zapytanie(int l, int r, T res = e◇) {
  for (l += leafs - 1, r += leafs - 1; l <= r; ) {
    if (l % 2 == 1) res = res ◇ tree[l++];
    if (r % 2 == 0) res = res ◇ tree[r--];
    l /= 2, r /= 2;
  }
  return res;
}
```

TODO