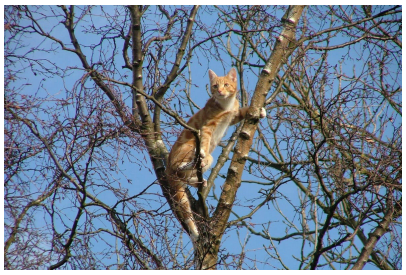
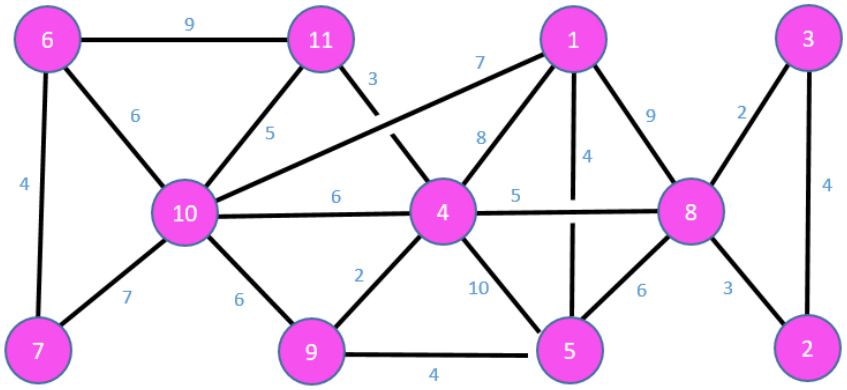
Minimalne drzewo rozpinające



#algorytm_Kruskala

Wyobraźmy sobie spójny graf nieskierowany posiadający V wierzchołków, w którym każdej krawędzi przypisaliśmy nieujemną *wagę* (ang. **weight**, wymawiaj: *łejt*), na przykład:



Naszym zadaniem jest wybranie takiego podzbioru krawędzi, aby:

- Tworzyły one drzewo zawierające wszystkie wierzchołki grafu,
- Suma wag tych krawędzi była minimalna.

Takie drzewo nosi nazwę *minimalnego drzewa rozpinającego* (ang. **MST**, **Minimum Spanning Tree**, wymawiaj: *minimum spaning tri*, z akcentem na pierwszą sylabę w pierwszym słowie oraz długim i na końcu.). Takie drzewa będzie miało dokładnie $V - 1$ krawędzi (dlaczego?).

Reprezentacja grafu ważonego

Dane wejściowe dla naszego grafu podamy w następującym formacie:

$V\ E$
$a_0\ b_0\ w_0$
$a_1\ b_1\ w_1$
$a_2\ b_2\ w_2$
$\ldots$
$a_{E-1}\ b_{E-1}\ w_{E-1}$

gdzie w_i, $i = 0, 1, 2, \ldots, E - 1$ są wagami poszczególnych krawędzi.

W naszym przykładzie może wyglądać to tak:

```
11 20
6 11 9
6 10 6
6 4 7
10 11 5
10 1 7
10 4 6
10 9 6
7 10 7
9 4 2
11 4 3
4 1 8
4 5 10
1 5 4
1 8 9
8 5 6
4 8 5
8 3 2
8 2 3
2 3 4
9 5 4
```

Do wczytania danych zaangażujemy specjalną funkcję `read_weighted_graph()`:

```
void read_weighted_graph(vector<vector<int>> &G, int &V, int &E,
                         vector<edge> &W)
{
  . . .
}
```

Mamy tu dwie nowinki: dodatkowy parametr W i nowy typ danych `edge`. Wektor W posłuży nam do przechowania krawędzi grafu. Typ danych `edge` to struktura, która reprezentować będzie pojedynczą krawędź łączącą wierzchołki a, b i posiadającą wagę w:

```
struct edge{
  int a, b, w;

  edge(int aa, int bb, int ww)
  {
    a = aa; b = bb; w = ww;
  }

  edge()
  {}
};
```

Przy okazji zdefiniowaliśmy dwa konstruktory dla tej struktury: trójparametrowy oraz bezparametrowy.

Czas na treść funkcji (do wstawienia zamiast trzech kropek):

```
cin >> V >> E;
G.resize(V + 1);
W.resize(E);
for(int i = 0; i < E; i++)
{
  int a, b, w;
  cin >> a >> b >> w;
  W[i] = edge(a, b, w);
  G[a].push_back(b);
  G[b].push_back(a);
}                                  }
```

Listy sąsiedztwa w grafie G budujemy tradycyjnie, pamiętając że jest to graf nieskierowany.

Początkowy fragment funkcji `main()` powinien wyglądać następująco:

```
int main()
{
  int V, E;
  vector<vector<int>> G;
  vector<edge> W;
  read_weighted_graph(G, V, E, W);
  vector<edge> MST_edges(V - 1);
  sort(W.begin(), W.end(), lighter);
  . . .
}
```

Ok, dane mamy wczytane, a ponadto zadeklarowaliśmy wektor *MST_edges*, w którym przechowamy wybrane krawędzie – elementy drzewa.

I co teraz? Algorytm, który zaproponujemy Czytelnikowi, jest algorytmem zachłannym, zwanym algorytmem Kruskala.* Wybór krawędzi przeznaczonych do minimalnego drzewa rozpinającego powinien zacząć się od „najlżejszej" krawędzi i na każdym kolejnym kroku powinniśmy wybierać dostępną (akceptowalną) krawędź o najmniejszej wadze.

W tym celu dokonamy sortowania wektora W, a do porównywania krawędzi posłuży nam funkcja-komparator `lighter()` (por. podrozdział *Kto lepszy?*):

```
bool lighter(const edge &v, const edge &u)
{
  return v.w < u.w;
}
```

Zatem kolejna linijka w funkcji `main()` to:

```
sort(W.begin(), W.end(), lighter);
```

*Można pokazać, że w tym zagadnieniu metoda zachłanna da poprawne rezultaty.

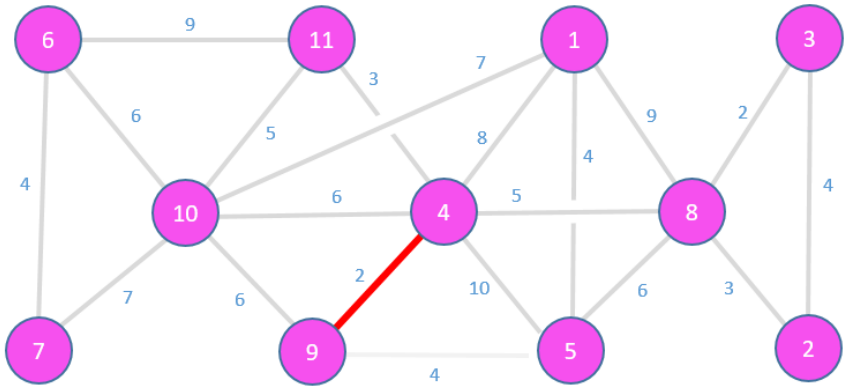
Kilka początkowych krawędzi w posortowanym wektorze to:

$(8, 3)$, $(9, 4)$, $(11, 4)$, $(8, 2)$, $(6, 7)$, $(9, 5)$, $(2, 3)$, ...

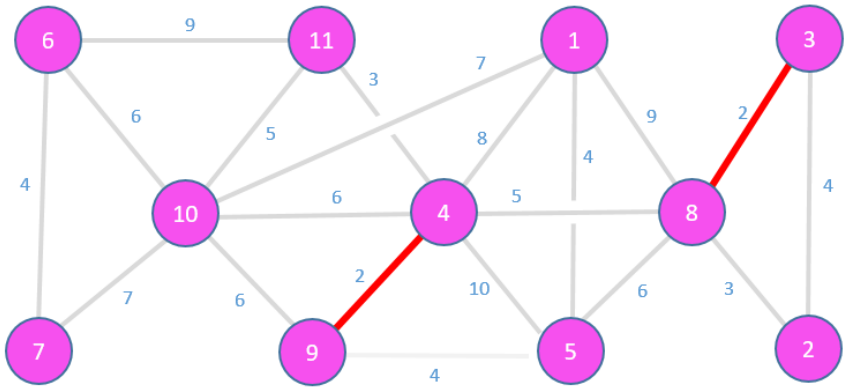
O ile na początku będzie szło gładko (zwłaszcza pierwsze dwie krawędzie), o tyle za niedługo zaczną się problemy: nie możemy bowiem bezkrytycznie brać kolejnej najlżejszej krawędzi. Musi ona być elementem przyszłego drzewa, a więc nie może tworzyć cyklu z krawędziami wybranymi do tej pory.

Struktura zbiorów rozłącznych

Pierwsza wzięta krawędź (najlżejsza w całym grafie) to $(9, 4)$ (o wadze 2):[†]

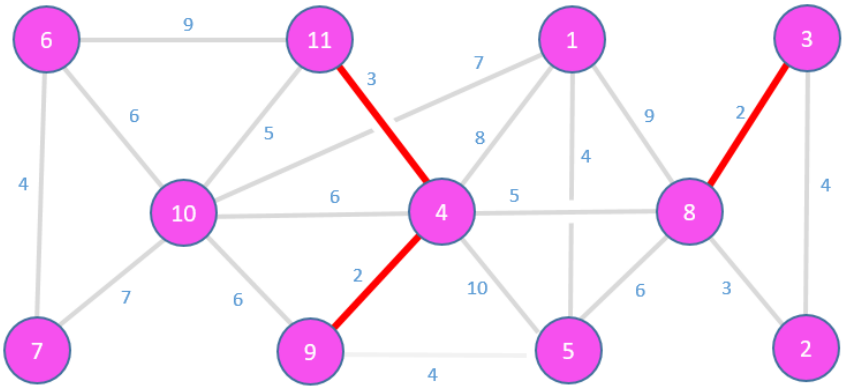


Kolejną krawędzią jest $(8, 3)$:

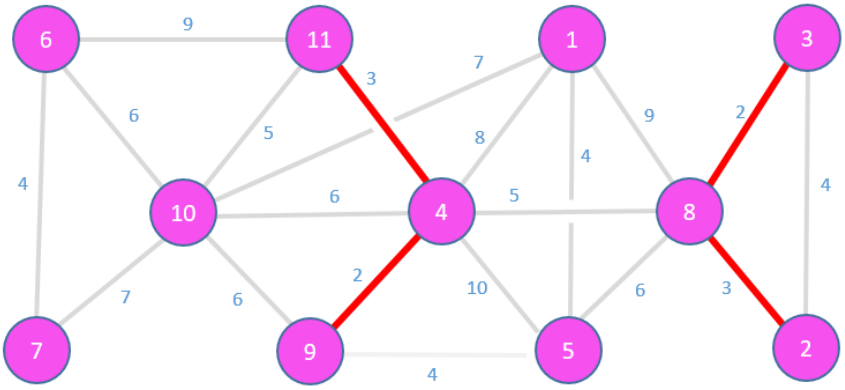


[†]W naszym grafie jest jeszcze jedna krawędź o takiej samej wadze $(8, 3)$ – w naszym przypadku wystąpi ona jako druga, bo tak wypadło w sortowaniu.

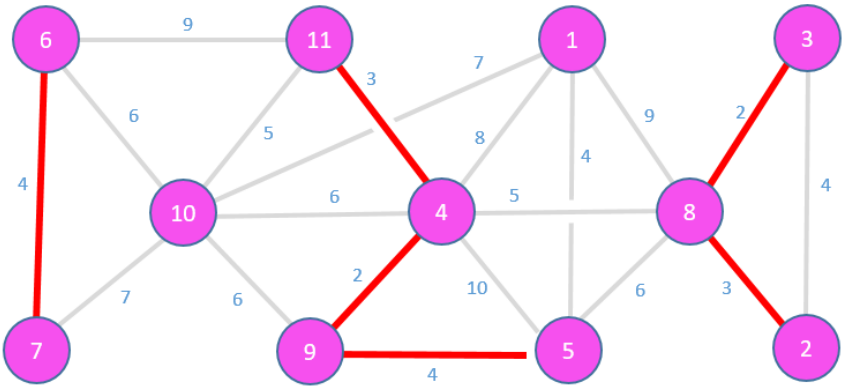
Teraz krawędź $(11, 4)$:



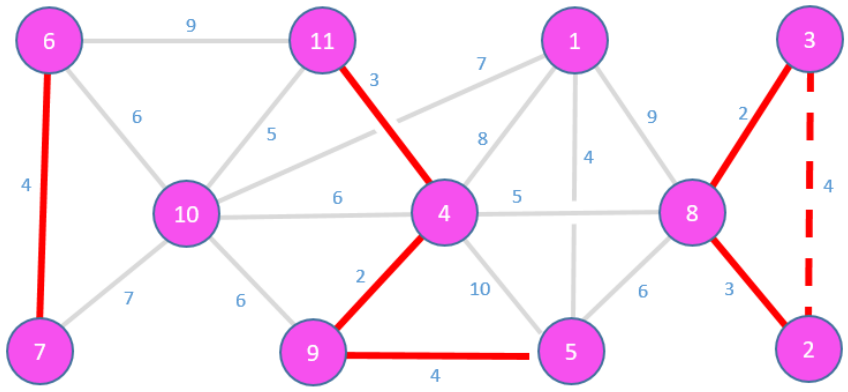
Krawędź $(8, 2)$:



Teraz kolejno krawędzie $(6, 7)$ oraz $(9, 5)$ (jedna po drugiej, aby nie przedłużać):



Aliści teraz czas na krawędź $(2, 3)$, której zaakceptować nie możemy, gdyż wraz z wybranymi już krawędziami $(8, 2)$ oraz $(8, 3)$ utworzyłaby cykl (niedopuszczalny w drzewie):



Jak szybko sprawdzić, czy nowa krawędź nie wygeneruje cyklu? Otóż mamy gotowe narzędzie – jest nim metoda **find-union** do operacji na zbiorach rozłącznych.

Na początku nie mamy żadnych krawędzi, więc każdy wierzchołek należy do oddzielnego (jednoelementowego) zbioru krawędzi. Dodanie krawędzi oznacza połączenie wierzchołków w jeden zbiór (**union**). Przed każdym następnym dodaniem należy sprawdzić, czy wierzchołki należą do różnych zbiorów – wtedy operacja **union** je połączy, w przeciwnym razie krawędź odrzucamy, bo utworzyłaby cykl. Tak jest właśnie z niedawno rozważaną krawędzią $(2, 3)$, która połączyłaby wierzchołki należące do jednego zbioru $\{2, 3, 8\}$.

Do naszego programu musimy wstawić definicje funkcji `make_set()`, `find_set()` oraz `union_set()`, które znamy z podrozdziału *Zbiory rozłączne*.

Możemy teraz napisać funkcję `MST()`, która zrealizuje cały algorytm Kruskala. Oto jej początek:

```
void MST(int V, int E, vector<edge> &W, vector<edge> &MST_edges)
{
  vector<int> MST_parent(V + 1), MST_rank(V + 1);
  make_set(MST_parent, MST_rank);
  . . .
}
```

Nazwy wektorów potrzebnych do funkcjonowania metody **find-union** zostały wzbogacone o przedrostek MST, aby nie myliły się z innymi nazwami z algorytmów grafowych. Poza tym zainicjalizowaliśmy strukturę rozłącznych zbiorów wierzchołków przy uzyciu funkcji `make_set()`.

Obecnie uruchomimy pętlę, która „pochyli się” nad każdą krawędzią (są one już posortowane niemalejąco według wag) i zbada, czy daną krawędź można dołączyć do wektora MST_edges na pozycji i:

```
int i = 0;
for(edge z : W)
{
  int a = z.a, b = z.b;
  int a_root = find_set(MST_parent, a);
  int b_root = find_set(MST_parent, b);
```

```
  if(a_root != b_root)
  {
    union_set(MST_parent, MST_rank, a_root, b_root);
    MST_edges[i++] = z;
  }
}
```

Zmienne *a_root* oraz *b_root* oznaczają reprezentantów zbiorów wierzchołków, do których należą końce rozważanej krawędzi. (Ten kod wstawiamy zamiast trzech kropek w funkcji `MST()`).

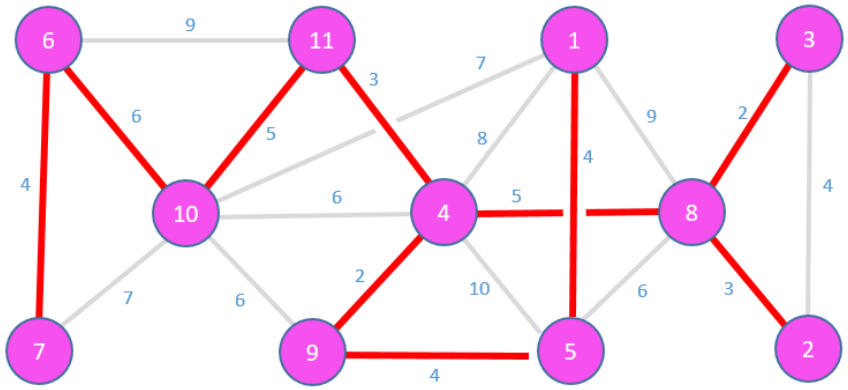
Jeszcze przytoczymy kompletny kod funkcji `main()` (z wypisaniem listy wybranych krawędzi):

```
int main()
{
  int V, E;
  vector<vector<int>> G;
  vector<edge> W;
  read_weighted_graph(G, V, E, W);
  sort(W.begin(), W.end(), lighter);
  vector<edge> MST_edges(V - 1);
  MST(V, E, W, MST_edges);
  for(edge z : MST_edges)
    cout << z.a << ' ' << z.b << endl;
}
```

Na ekranie zobaczymy listę krawędzi tworzących minimalne drzewo rozpinające:

```
9 4
8 3
11 4
8 2
6 7
9 5
1 5
10 11
4 8
6 10
```

A tak prezentuje się to drzewo w pełnej okazałości:



Warto podkreślić, że drzewo **MST** nie musi być określone jednoznacznie, natomiast waga drzewa minimalnego jest zawsze ściśle określona (przerabialiśmy już podobne klimaty).