Spójność i kolorowanie grafu

#spójna_składowa
#dwukolorowanie

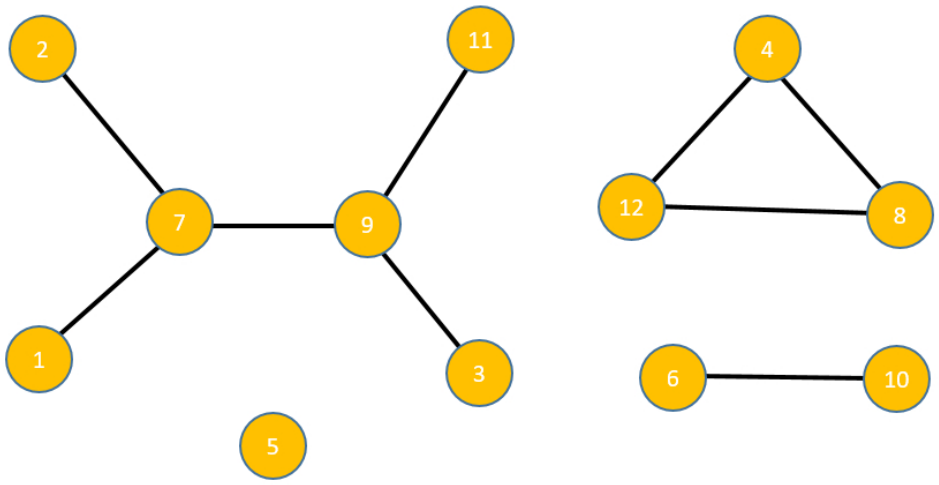
Algorytm BFS może być pomocny w bardzo wielu sytuacjach, a także może być elementem lub punktem wyjścia w bardziej zaawansowanych algorytmach grafowych.

W pierwszej sekcji opiszemy sposób znajdowania ilości spójnych składowych grafu nieskierowanego, a w drugiej – sprawdzenie, czy wierzchołki grafu da się pokolorować tak, aby wierzchołki o tym samym kolorze nie były bezpośrednimi sąsiadami.

Spójne składowe grafu

Dla grafu nieskierowanego spójność oznacza tyle, że z każdego wierzchołka można się dostać do każdego innego, być może przechodząc przez więcej niż jedną krawędź.*

Łatwo wyobrazić sobie graf, który nie spełnia tego warunku, na przykład:



Jak widać, samotny wierzchołek (tutaj: 5) jest pełnoprawnym elementem grafu. Nawiasem mówiąc, w grafach niespójnych nie obowiązuje reguła $E \geqslant V - 1$, zachodzi raczej $E \geqslant 0$.

Dane wejściowe dla powyższego grafu mogą wyglądać następująco:

```
12 9
2 7
```

*Dla grafu skierowanego definicja jest w zasadzie taka sama, z tym że wtedy mówimy o *silnej spójności*. Temu ważnemu zagadnieniu poświęcimy oddzielny podrozdział w tych *Miaukotach*.

```
7 1
6 10
12 8
7 9
9 11
9 3
4 8
4 12
```

Funkcja `print_graph()` zaproponuje nam następujące listy sąsiedztwa:

```
1: 7
2: 7
3: 9
4: 8 12
5:
6: 10
7: 2 1 9
8: 12 4
9: 7 11 3
10: 6
11: 9
12: 8 4
```

Lista sąsiedztwa dla wierzchołka 5 jest pusta – zgodnie z oczekiwaniami.

Graf, który nie jest spójny jako całość, możemy podzielić na *spójne składowe* (ang. **connected components**, wymawiaj: *konekted komponents*, w drugim wyrazie z akcentem na pierwszą sylabę). Oczywiście spójne składowe są odizolowane od siebie: na rysunku przedstawiającym nasz graf na oko widzimy 4 takie składowe: $\{1, 2, 3, 7, 9, 11\}$, $\{4, 8, 12\}$, $\{6, 10\}$ oraz $\{5\}$.

Napiszemy teraz program, który wyznaczy ilość spójnych składowych grafu oraz zakwalifikuje wierzchołki grafu do poszczególnych składowych. W tym celu zmodyfikujemy nieco funkcję `BFS()` – zresztą nazwiemy tę nową wersję `BFS_connect()`.

Zaczniemy od nagłówka funkcji – wektor *visited* będzie miał od tej pory elementy typu `int` (będziemy tu zapisywać numer spójnej składowej, do której należy dany wierzchołek – wartość 0 oznacza, że nie był on jeszcze odwiedzony i zakwalifikowany), a ponadto dodamy argument *component* oznaczający bieżący numer spójnej składowej:

```
void BFS_connect(vector<vector<int>> G, int V, int s,
         vector<int> &visited, int component,
         vector<int> &parent, vector<int> &dist)
{
  . . .
```

Nie usuwaliśmy parametrów *parent* oraz *dist*, mimo że nie będą nam jawnie potrzebne w tym problemie.

Zmienna *component* będzie miała nadana wartość w funkcji `main()` (lub innej funkcji, która wywoła funkcję `BFS_connect()`).

Odwiedzamy wierzchołek startowy i wstawiamy go do kolejki:

```
visited[s] = component;
dist[s] = 0;
queue<int> Q;
Q.push(s);
```

Teraz uruchamiamy pętlę przetwarzającą kolejkę Q:

```
while(!Q.empty())
  {
    int v = Q.front();
    Q.pop();
    for(int x : G[v])
      if(!visited[x])
      {
        visited[x] = component;
        parent[x] = v;
        dist[x] = dist[v] + 1;
        Q.push(x);
      }
  }
}
```

Właściwie jedyną nowością w kodzie funkcji jest użycie zmiennej *component* zamiast wartości `true`. Wartość 0 jest traktowana jako `false`, więc nie trzeba zmieniać warunku w instrukcji warunkowej.

Dopiszmy jeszcze funkcję `print_BFS_connect()` i dopasowaną do problemu funkcję `main()`. Tym razem będziemy starali się uruchomić przeszukiwanie kolejno z każdego wierzchołka (w funkcji `main()`) – ma to sens, jeśli wierzchołek jest nieodwiedzony. Jeżeli tak w istocie będzie, wtedy zwiększamy numer spójnej składowej i wywołujemy funkcję `BFS_connect()`:

```
void print_BFS_connect(vector<vector<int>> G, int V, vector<int> visited)
{
  for(int v = 1; v <= V; v++)
    cout << v << ": " << visited[v] << endl;
}

int main()
{
  int V, E;
  vector<vector<int>> G;
  read_graph(G, V, E);
  vector<int> visited(V + 1), parent(V + 1), dist(V + 1);
  int component = 0;
  for(int v = 1; v <= V; v++)
    if(!visited[v])
    {
```

```
      component++;
      BFS_connect(G, V, v, visited, component, parent, dist);
    }
  cout << component << endl;
  print_BFS_connect(G, V, visited);
}
```

Po uruchomieniu programu i wpisaniu danych przedstawionych w początku tej sekcji otrzymamy wydruk:

```
4
1: 1
2: 1
3: 1
4: 2
5: 3
6: 4
7: 1
8: 2
9: 1
10: 4
11: 1
12: 2
```

Pojedyncza liczba 4 wypisana na początku oznacza ilość spójnych składowych grafu.

Kolorowanie grafu

Przypisanie kolorów poszczególnym wierzchołkom grafu według określonych reguł jest w ogólnym przypadku dość zawiłym problemem, zatem ograniczymy się tylko do napisania programu, który odpowie na pytanie, czy możliwe jest *dwukolorowanie* danego grafu, czyli pokolorowanie dwoma kolorami, aby wierzchołki bezpośrednio połączone krawędzią miały zawsze różne kolory. Zatem program powinien wypisać np. **TAK**, gdy jest to możliwe, lub **NIE** – w przeciwnym przypadku.

Przyda nam się znowu wektor *visited* o elementach całkowitych: możemy przyjąć że wartości ± 1 odpowiadają różnym kolorom, natomiast wartość 0 oznacza jak poprzednio wierzchołek nieodwiedzony.

Tym razem może zaczniemy od funkcji `main()`:

```
int main()
{
  int V, E;
  vector<vector<int>> G;
  read_graph(G, V, E);
  if(two_colors(G, V))
    cout << "TAK" << endl;
```

```
  else
    cout << "NIE" << endl;
}
```

No to teraz funkcja `two_colors()`, która uruchamia przeszukiwanie wszerz dla kolejnych spójnych składowych (każda z nich będzie badana niezależnie):

```
bool two_colors(vector<vector<int>> &G, int V)
{
  vector<int> visited(V + 1);
  for(int v = 1; v <= V; v++)
    if(!visited[v])
    {
      if(!BFS_color(G, V, v, visited)
        return false;
    }
  return true;
}
```

Użycie nawiasów klamrowych otaczających wewnętrzną instrukcję warunkową nie jest konieczne, ale – naszym zdaniem – jest dobrą praktyką prewencyjną (przed uniknięciem bardzo nieoczywistych i trudnych do wykrycia błędów przy złożonych konstrukcjach logicznych).

Funkcja `BFS_color()` nie musi wyznaczać poprzedników ani odległości, więc pominiemy niepotrzebne parametry i nieco uprościmy pętlę przeszukującą. Jeśli dla choć jednej składowej grafu nie da się ustalić kolorów, wtedy wartość całej funkcji to `false`. Jeżeli natomiast pętla po v obejdzie spokojnie do końca, wtedy zwracamy wartość `true`.[†]

Przyszedł czas na funkcję `BFS_color()`, czyli nieco okrojoną wersję funkcji `BFS()`. Zaczynamy tradycyjnie plus to, że ustalamy bieżącą wartość koloru. (Kolor przełączymy na ten drugi przez zmianę znaku.)

```
bool BFS_color(vector<vector<int>> G, int V, int s, vector<int> &visited)
{
  int color = 1;
  visited[s] = color;
  queue<int> Q;
  Q.push(s);
```

Następnie przychodzi pętla przetwarzającą kolejkę Q:

```
while(!Q.empty())
  {
    int v = Q.front();
    Q.pop();
    for(int x : G[v])
      if(!visited[x])
      {
```

[†]Niezmienny szacunek dla profesora de Morgana!

```
            visited[x] = -visited[v]; // przeciwny kolor
            Q.push(x);
          }
          else // już pokolorowany wierzchołek
            if(visited[x] == visited[v])
              return false;
        }
        return true;
      }
```

Jeśli napotkamy obok siebie dwa pokolorowane wierzchołki, wtedy sprawdzamy, czy ich kolory są takie same. Jeśli tak jest, pokolorowanie jest niemożliwe. Jeżeli natomiast pętla `while` obejdzie do końca, wtedy zwracamy wartość `true` (znów prawa de Morgana).

Dla naszego przykładowego grafu zaprezentowanego na początku tego podrozdziału otrzymamy odpowiedź negatywną – nie jest możliwe dwukolorowanie grafu, który zawiera cykl o nieprzystej liczbie krawędzi (dlaczego?).