# Klasa Graph

Graph jest abstrakcyjną klasą bazową dla wszystkich klas implementujących grafy. Biblioteka oferuje dwie klasy pochodne:

- AdjacencyMatrixGraph – graf reprezentowany za pomocą macierzy sąsiedztwa.
- AdjacencyListsGraph<AL> – graf reprezentowany za pomocą list sąsiedztwa. Klasa jest parametryzowana typem AL implementującym interfejs IAdjacencyList, który jest używany jako słownik przechowujący sąsiadów wierzchołków. Biblioteka oferuje trzy implementacje tego interfejsu:
  - SimpleAdjacencyList – używa zwykłej listy,
  - AVLAdjacencyList – używa drzew AVL,
  - HashTableAdjacencyList – używa tablicy haszowanej.

Pochodne klasy Graph mają dwa konstruktory:

- Konstruktor z jednym parametrem typu Graph – tworzy kopię grafu będącego parametrem,
- Konstruktor z dwoma parametrami typów bool i int – tworzy graf składający się ze wskazanej liczby izolowanych wierzchołków; parametr typu bool określa, czy tworzony jest graf skierowany (wartość true oznacza, że tak).

Przykład:

```
Graph G = new AdjacencyMatrixGraph(false, 15)
Graph G2 = new AdjacencyListsGraph<AVLAdjacencyList>(G)
```

Posługując się klasą Graph należy pamiętać o następujących konwencjach:

- Liczba wierzchołków oraz to, czy graf jest skierowany, nie zmienia się w trakcie życia obiektu.
- Wierzchołki grafu numerowane są kolejnymi liczbami całkowitymi, poczynając od 0.
- Wszystkie grafy są ważone (w problemach dla grafów nieważonych należy ignorować wagi krawędzi, a przy dodawaniu krawędzi – pozostawiać domyślną wagę 1).
- Działając na typie Graph należy obowiązkowo abstrahować od faktycznego typu obiektu (na przykład kiedy dostajemy jako argument obiekt typu Graph i chcemy działać na jego kopii, błędem jest wywołanie konstruktora konkretnej klasy – AdjacencyMatrixGraph lub AdjacencyListsGraph). Można natomiast użyć nastepujących metod klasy Graph:
  - Clone – metoda tworzy głęboką kopię bieżącego grafu (tego samego typu),
  - IsolatedVerticesGraph – metoda tworzy graf tego samego typu. W wersji bezparametrowej liczba wierzchołków i „skierowalność” pozostają takie same, w wersji z dwoma parametrami obie te właściwości można zmienić.

Kilka wskazówek odnośnie operacji na krawędziach:

- Dodawanie i usuwanie krawędzi realizujemy wywołując metody odpowiednio AddEdge i DelEdge. Metody zgłaszają wyjątek, gdy numery wierzchołków wychodzą poza zakres oraz zwracają false, gdy operacja nie może zostać wykonana (usuwanie nieistniejącej krawędzi, dodawanie już istniejącej krawędzi).
- Do sprawdzenia istnienia krawędzi i wagi krawędzi służy metoda GetEdgeWeight. Gdy krawędź nie istnieje w grafie, zwrócona zostanie wartość NaN (uwaga: porównanie z wartością NaN należy wykonywać używając metody IsNaN, bo operator == zawsze zwraca false).
- Wylistowanie wszystkich krawędzi wychodzacych z zadanego wierzchołka grafu najłatwiej (i najwydajniej) zrobić posługując się metodą OutEdges.

# Przeszukiwanie grafu

Biblioteka umożliwia przeszukiwanie grafu poczynając od zadanego wierzchołka zgodnie z następującym, ogólnym schematem.

**procedure** GeneralSearchFrom($G$: graf, $v_0 \in V(G)$)
    $K \leftarrow$ pusta kolekcja krawędzi
    Wstaw wszystkie krawędzie wychodzące z $v_0$ do $K$
    **while** $K$ jest niepusta **do**
        pobierz z kolekcji krawędź $xy$
        **if** wierzchołek $y$ jest nieodwiedzony **then**
            Oznacz $y$ jako odwiedzony
            Wstaw wszystkie krawędzie wychodzące z $y$ do $K$

Zauważmy, że działanie procedury jest zależne od typu kolekcji $K$ – na przykład kiedy $K$ jest kolejką, realizowane jest przeszukiwanie wszerz (BFS), a gdy $K$ jest stosem – przeszukiwanie w głąb (DFS).

Dostęp do tej funkcjonalności uzyskujemy za pośrednictwem metody GeneralSearchFrom<T> rozszerzającej interfejs Graph. Metoda przyjmuje następujące parametry:

- T – typ kolekcji $K$ używanej w przeszukiwaniu, implementujący interfejs IEdgesContainer. Przydatne implementacje:
  - EdgesStack – stos
  - EdgesQueue – kolejka
  - EdgesMinPriorityQueue, EdgesMaxPriorityQueue – kolejki priorytetowe
- from – wierzchołek, z którego rozpoczynamy poszukiwania.
- preVisitVertex – predykat (typu Predicate<Int32>) wywoływany w momencie oznaczania wierzchołka jako odwiedzony. Wartość zwracana jest interpretowana jako informacja, czy kontynuować przeszukiwanie.
- postVisitVertex – predykat wywoływany po przetworzeniu (usunięciu w $K$) wszystkich krawędzi wychodzacych z wierzchołka. Korzystanie z tego argumentu jest dozwolone jedynie gdy $K$ jest typu EdgesStack, czyli dla przeszukiwania w głąb (w innych przypadkach metoda zgłasza wyjątek ArgumentException).
- visitEdge – predykat (typu Predicate<Edge>) wywoływany dla każdej przetwarzanej krawędzi.
- visitedVertices – tablica typu bool[] z informacją, które wierzchołki zostaną pominęte przy przeszukiwaniu

Dodatkowo, bilioteka udostępnia metodę GeneralSearchAll, która realizuje przeszukiwanie całego grafu poprzez wielokrotne wywoływanie metody GeneralSearchFrom dla jeszcze nieodwiedzonego wierzchołka dopóki takie wierzchołki znajdują się w grafie. Parametry metody są analogiczne, poza następującymi różnicami:

- cc – parametr wyjściowy, informacja o liczbie wywołań metody GeneralSearchFrom
- nr – Tablica kolejności „wierzchołków startowych” (jako wierzchołek startowy dla kolejnego wywołania metody GeneralSearchFrom wybierany jest pierwszy nieodwiedzony wierzchołek nr[i], gdzie tablica nr przeglądana jest w kierunku rosnących indeksów)

# Zadanie: badanie dwudzielności i algorytm Kruskala

Uzupełnić w klasie Lab03GraphFunctions następujące metody:

- Graph Lab03Reverse(Graph g) – metoda wyznaczająca odwrotność zadanego grafu skierowanego, gdzie odwrotność grafu to graf skierowany o wszystkich krawędziach przeciwnie skierowanych niż w grafie pierwotnym.
- bool Lab03IsBipartite(Graph g, out int[] vert) – metoda sprawdzająca, czy zadany graf jest dwudzielny i, jeśli tak, zwracająca 2-kolorowanie za pośrednicwem parametru wyjściowego vert.
- Graph Lab03Kruskal(Graph g, out double mstw) – wyznaczanie minimalnego drzewa rozpinającego algorytmem Kruskala.
- bool Lab03IsUndirectedAcyclic(Graph g) – sprawdzenie, czy zadany graf jest acykliczny.

**Punktacja:**

- Etap 1. 0.5 punktu,
- Etap 2. 0.5 punktu,
- Etap 3. 1 punkt,
- Etap 4. 0.5 punkt.

**Wskazówki:**

- Drugą i czwarta częśc zadania najłatwiej rozwiązać wykorzystując metodę GeneralSearchFrom
- W implementacji algorytmu Kruskala przydatne mogą być klasy UnionFind oraz EdgesMinPriorityQueue z blibioteki Graph

# Zadanie: silny indeks chromatyczny

Zadanie składa się z czterech cześci; części 1-3 są niezależne od siebie, natomiast część 4 korzysta ze wszystkich poprzednich.

Uwagi do wszystkich metod:

1. Grafy wynikowe muszą być reprezentowane w taki sam sposób jak grafy będące parametrami
2. Grafów będących parametrami nie wolno zmieniać

**Część I: Funkcja zwracajaca kwadrat danego grafu**

Kwadratem grafu nazywamy graf o takim samym zbiorze wierzchołków jak graf pierwotny, w którym wierzchołki połączone sa krawędzią jeśli w grafie pierwotnym były połączone krawędzia bądź ścieżką złożoną z 2 krawędzi (ale pętli, czyli krawędzi o początku i końcu w tym samym wierzchołku, nie dodajemy!).

**Część II: Funkcja zwracająca Graf krawędziowy danego grafu**

Wierzchołki grafu krawędziowego odpowiadają krawędziom grafu pierwotnego, wierzchołki grafu krawędziowego połączone są krawędzią jeśli w grafie pierwotnym z krawędzi odpowiadającej pierwszemu z nich można przejść na krawędź odpowiadającą drugiemu z nich przez wspólny wierzchołek (jeśli graf pierwotny jest skierowany to wierzchołki grafu krawędziowego połączone są krawędzią jeśli wierzchołek końcowy krawędzi odpowiadajacej pierwszemu z nich jest wierzchołkiem początkowym krawędzi odpowiadajęcej drugiemu z nich).

Tablicę names tworzymy i wypełniamy według następującej zasady: każdemu wierzchołkowi grafu krawędziowego odpowiada element tablicy names (o indeksie równym numerowi wierzchołka) zawierający informację z jakiej krawędzi grafu pierwotnego wierzchołek ten powstał, np. dla wierzchołka powstałego z krawedzi <0,1> do tablicy zapisujemy krotkę (0, 1) - przyda się w dalszych etapach.

UWAGA: Graf pierwotny może być skierowany lub nieskierowany, graf krawędziowy zawsze jest nieskierowany.

**Część III: Funkcja znajdujaca poprawne kolorowanie wierzchołków danego grafu nieskierowanego**

Kolorowanie wierzchołków jest poprawne, gdy każde dwa sąsiadujące wierzchołki mają różne kolory.

Funkcja ma szukać kolorowania według następujacego algorytmu zachłannego: Dla wszystkich wierzchołków $v$ (od 0 do $n-1$) pokoloruj wierzcholek $v$ kolorem o najmniejszym możliwym numerze (czyli takim, na który nie są pomalowani jego sąsiedzi).

Kolory numerujemy począwszy od 0.

Funkcja zwraca liczbę użytych kolorów (czyli najwyższy numer użytego koloru + 1), a w tablicy colors zapamiętuje kolory poszczególnych wierzchołków.

UWAGA: Podany opis wyznacza kolorowanie jednoznacznie, jakiekolwiek inne kolorowanie, nawet jeśli spełnia formalnie definicję kolorowania poprawnego, na potrzeby tego zadania będzie uznane za błędne.

UWAGA 2: Dla grafów skierowanych metoda powinna zgłaszać wyjątek ArgumentException.

**Część IV: Funkcja znajduje silne kolorowanie krawędzi danego grafu**

Silne kolorowanie krawędzi grafu jest poprawne gdy każde dwie krawędzie, które są ze sobą sąsiednie (czyli można przejść z jednej na drugą przez wspólny wierzchołek, uwzględniając kierunek krawędzi) albo są połączone inną krawędzią (czyli można przejść z jednej na drugą przez ową inną krawędź, uwzględniając kierunek wszystkich biorących w tym udział krawędzi), mają różne kolory.

Należy zwrocić nowy graf, który będzie miał strukturę identyczną jak zadany graf, ale w wagach krawędzi zostaną zapisane przydzielone kolory.

Wskazówka: to bardzo proste. Należy wykorzystać wszystkie poprzednie funkcje. Zastanowić się co możemy powiedzieć o kolorowaniu wierzchołków kwadratu grafu krawędziowego? Jak się to ma do silnego kolorowania krawędzi grafu pierwotnego?