

1 Implementacja algorytmu Fleury'ego jako przykład wykorzystujący koncepty

1.1 Omówienie problemu

Algorytm Fleury'ego to algorytm pozwalający na znalezienie *cyklu Eulera* w grafie eulerowskim.

Definicja 1. Graf - struktura służąca do przedstawiania i badania relacji między obiektami. Jest to zbiór wierzchołków, które mogą być połączone krawędziami, gdzie krawędź zaczyna się i kończy w którymś z wierzchołków.

Definicja 2. Multigraf - graf, w którym mogą występować krawędzie wielokrotne (powtarzające się) oraz pętle (krawędzie, których końcami jest ten sam wierzchołek).

Definicja 3. Graf spójny - graf spełniający warunek, że dla każdej pary wierzchołków istnieje ścieżka, która je łączy.

Definicja 4. Ścieżka - ciąg wierzchołków, połączonych krawędziami.

Definicja 5. Droga - ścieżka, w której wierzchołki są różne.

Definicja 6. Cykl - droga zamknięta czyli ścieżka, w której pierwszy i ostatni wierzchołek są równe.

Definicja 7. Graf eulerowski - spójny multigraf posiadający cykl, który zawiera wszystkie krawędzie.

Definicja 8. Warunek istnienia cyklu Eulera w spójnym multigrafie - stopień każdego wierzchołka musi być liczbą parzystą.

Data: $G = (V, E)$, G - spójny multigraf, V - zbiór wierzchołków, E - zbiór krawędzi

Result: zbiór wierzchołków reprezentujących cykl Eulera

Zaczynamy od dowolnego wierzchołka ze zbioru V ;

```
while Dopóki zbiór krawędzi nie jest pusty do
    if Jeżeli z bieżącego wierzchołka  $x$  odchodzi tylko jedna krawędź
        then
            to przechodzimy wzdłuż tej krawędzi do następnego
            wierzchołka i usuwamy tę krawędź wraz z wierzchołkiem  $x$ ;
        else
            wybieramy tę krawędź, której usunięcie nie rozspójnia grafu i
            przechodzimy wzdłuż tej krawędzi do następnego
            wierzchołka, a następnie usuwamy tę krawędź z grafu;
        end
end
```

end

Algorytm 1: Algorytm Fleury'ego

1.2 Działanie programu

Założeniem programu jest symulacja algorytmu Fleury’ego dla jak największej ilości kontenerów biblioteki STL. Dzięki przeciążaniu funkcji jakie oferują koncepty, w prosty i czytelny sposób, udało się napisać generyczny algorytm.

W programie są dwa kontenery do przechowywania krawędzi i wierzchołków. Krawędź reprezentowana jest przez klasę `Edge`, która przyjmuje dwie wartości typu `int` do konstruktora. Wierzchołek, z kolei reprezentowany jest przez zmienną `int`.

Dane (pary wierzchołków) są wczytywane z pliku, a potem w zależności od rodzaju kontenera i iteratora, sortowane. Dla kontenerów:

- sekwencyjnych z iteratorem `Random access (vector)` wywoływana jest funkcja szablonu ograniczonego przez koncepty: `Sequence` i `Random_access_iterator`.

```
template<Sequence S, Random_access_iterator R>
void sortVertices(S &seq){
    sort(seq.begin(), seq.end());
}
```

- sekwencyjnych z iteratorem `Bidirectional (list)` lub sekwencyjnych `forward(forward_list)` z iteratorem `Forward` wywoływana jest funkcja szablonu ograniczonego przez koncepty: `Sequence` i `Bidirectional_iterator` lub `SequenceForward` i `Forward_iterator`

```
template<typename S, typename R>
requires (Sequence<S>() &&
Bidirectional_iterator<R>()) ||
(SequenceForward<S>() && Forward_iterator<R>())
void sortVertices(S &seq){
    seq.sort();
}
```

- asocjacyjnych (`set`) wywoływana jest funkcja szablonu ograniczonego przez koncept: `Associative_container`.

```
template<Associative_container A>
void sortVertices(A &seq){}
```

Omawiany algorytm wykonuje funkcja `determineEulerCycle`:

```
template<typename E, typename V>
void determineEulerCycle(E &edges, V &vertices){
    int v = 0;
    bool condition = (checkIfGraphConnected(edges,
        vertices, 0, v) && checkIfAllEdgesEvenDegree
        (edges, vertices));

    if(condition){
        cout << "Euler_cycle:" << endl << endl << v;
        while(!edges.empty()){
            switch(getNeighboursCount(edges, v)){
                case 1 : {
                    removeEdgeWithOneNeighbour(edges, v);
                    break;
                }
                default: {
                    removeEdgeWithMoreNeighbour(edges,
                        v, vertices);
                    break;
                }
            }
            cout << " -> " << v;
        }
        cout << endl << endl;
    } else {
        cout << "Invalid_graph." << endl;
        if(!checkIfGraphConnected(edges, vertices,
            0, v))
            cout << "Graph_is_not_connected" << endl;
        else if(!checkIfAllEdgesEvenDegree(edges,
            vertices))
            cout << "Not_all_the_edges_are_even" << endl;
    }
}
```

Żeby algorytm się wykonał, muszą zostać spełnione dwa warunki: graf musi być spójny (za to odpowiedzialna jest funkcja `checkIfGraphConnected`) i wszystkie krawędzie muszą być parzystego stopnia (`checkIfAllEdgesEvenDegree`).
`checkIfGraphConnected()`

```
template<typename E, typename V>
bool checkIfGraphConnected(E &ed, V &vertices,
int x, int startVertice) {
```

```

    bool *visited = new bool[vertices.size()];
    for (int i = 0; i < vertices.size(); i++)
        visited[i] = false;

    stack<int>stack;
    int vc = 0;

    stack.push(startVertice);
    visited[startVertice] = true;

    while (!stack.empty()) {
        int v = stack.top();
        stack.pop();
        vc++;

        for(typename E::iterator it = ed.begin();
            it != ed.end(); it++){
            if(it->getA() == v && !visited[it->getB()]){
                visited[it->getB()] = true;
                stack.push(it->getB());
            } else if(it->getB() == v &&
                !visited[it->getA()]){
                visited[it->getA()] = true;
                stack.push(it->getA());
            }
        }
    }

    delete [] visited;

    return (vc == vertices.size()-x);
}

```

Algorytm przechodzi przez graf, po kolei wrzuca odwiedzane wierzchołki na stos, zaznacza je w tablicy odwiedzonych (**visited**) i zaraz zdejmuje z tego stosu, zwiększając licznik **vc**. Robi to dopóki stos nie jest pusty. Zwraca warunek porównujący licznik **vc** z rozmiarem kontenera wierzchołków (wszystkie wierzchołki zostały odwiedzone, czyli istnieją ścieżki między wierzchołkami, graf jest spójny).

checkIfAllEdgesEvenDegree:

```

template<typename E, typename V>
bool checkIfAllEdgesEvenDegree(E &edges, V &vertices){

```

```

    int counter = 0, i = 0;
    for(auto v : vertices){
        for(auto e : edges){
            if(e.getA() == v || e.getB() == v)
                counter++;
        }
        if(counter % 2 == 0) i++;
    }
    return (i == vertices.size()) ? true : false;
}

```

Zmienna `i` zwiększa się jeśli ilość wystąpień wierzchołka jest liczbą parzystą. Zwraca wartość `true` jeśli zmienna `i` jest równa liczbie elementów kontenera zawierającego wierzchołki (dla każdego wierzchołka zmienna `i` zwiększała się o 1).

Jeśli warunek nie zostanie spełniony, użytkownik zostaje poinformowany o tym, że graf jest niepoprawny. W odwrotnej sytuacji, w pętli (dopóki kontener krawędzi nie jest pusty), wykonywana jest jedna dwóch operacji. Gdy wierzchołek ma jednego sąsiada, wywołuje się funkcja `removeEdgeWithOneNeighbour()`, a gdy więcej wierzchołków, funkcja `removeEdgeWithMoreNeighbour()`. Pierwsza z nich ma dwa przeciążenia konceptowe:

- Dla kontenera sekwencyjnego lub kontenera sekwencyjnego *forward*

```

template<typename S> requires Sequence<S>()
|| SequenceForward<S>()
void removeEdgeWithOneNeighbour(S &edges, int &v){

    typename S::iterator it =
        findElement<S>(edges, v);

    if(it->getA() == v) v = it->getB();
    else v = it->getA();

    deleteElementFromContainer(edges, it);
}

```

- Dla kontenera asocjacyjnego:

```

template<Associative_container A>
void removeEdgeWithOneNeighbour(A &edges, int &v){

    typename A::iterator it2;
    for(typename A::iterator it = edges.begin();
        it != edges.end(); it++){

```

```

        if(it->getA() == v || it->getB() == v){
            it2 = it;
            if(it->getA() == v) v = it->getB();
            else v = it->getA();
            it = prev(edges.end());
        }
    }

    edges.erase(it2);
}

```

Funkcja znajduje krawędź, dostając wierzchołek wychodzący. I wierzchołek znalezionej krawędzi przypisuje do tego przekazanego.

Druga `removeEdgeWithMoreNeighbour()` wygląda:

```

template<typename E, typename V>
void removeEdgeWithMoreNeighbour(E &edges, int &v,
V &vertices){

    bool w = true;

    for(typename E::iterator i = edges.begin();
i != edges.end() && w; i++){
        if(i->getA() == v && checkIfStillConnected
(edges, *i, getZeroDegreeCount(edges,
vertices), v, vertices)){
            v = i->getB();
            deleteElementFromContainer(edges, i);
            w = false;
        } else if(i->getB() == v &&
checkIfStillConnected(edges,
*i, getZeroDegreeCount(edges, vertices), v,
vertices)){
            v = i->getA();
            deleteElementFromContainer(edges, i);
            w = false;
        }
    }
}

```

Jeśli wierzchołek ma więcej sąsiadów, wybiera tego który nie rozspójni grafu. Żeby to sprawdzić używa funkcji `checkIfStillConnected`:

```
template<typename E, typename V>
bool checkIfStillConnected(E &edges, Edge e, int x,
int startVertice, V &vertices){

    E tmp;

    for(auto e : edges)
        addElementToContainer(tmp, e);

    bool w = true;
    for(typename E::iterator it = tmp.begin();
it != tmp.end() && w; it++){
        if (it->getA() == e.getA() &&
it->getB() == e.getB()) {
            deleteElementFromContainer(tmp, it);
            w = false;
        }

    }

    return checkIfGraphConnected(tmp, vertices,
x, startVertice);
}
```

W celu sprawdzenia, czy graf po usunięciu jakiejś krawędzi dalej będzie spójny, potrzebny jest pomocniczy kontener. Zapisujemy do niego aktualne krawędzie, wyszukujemy w nim przekazaną krawędź i przekazujemy go do istniejącej już funkcji `checkIfGraphConnected()`.

Dodatkowe funkcje:

`addElementToContainer`:

Dla kontenera sekwencyjnego, ograniczona przez koncept `Sequence`:

```
template<Sequence S, typename Element>
void addElementToContainer(S &seq, Element e){
    seq.push_back(e);
}
```

Dla kontenera sekwencyjnego *forward*, ograniczona przez koncept `SequenceForward`:

```
template<SequenceForward S, typename Element>
void addElementToContainer(S &seq, Element e){
    seq.push_front(e);
}
```

Dla kontenera asocjacyjnego, ograniczona przez koncept `Associative_container`:

```
template<Associative_container S, typename Element>
void addElementToContainer(S &seq, Element e){
    seq.insert(e);
}
```

Funkcja dodaje element do kontenera.

`findElement`:

Dla kontenera sekwencyjnego lub sekwencyjnego *forward*, ograniczona przez koncept `Sequence` lub `SequenceForward`

```
template<typename S>
requires Sequence<S>() || SequenceForward<S>()
typename S::iterator findElement(S &seq, int e){
    return find(seq.begin(), seq.end(), e);
}
```

Funkcja zwraca znaleziony element w kontenerze

`getVerticesSize`:

```
template<typename S>
int getVerticesSize(typename S::iterator begin,
typename S::iterator end){
    return distance(begin, end);
}
```

Funkcja zwraca rozmiar kontenera.

`deleteElementFromContainer`:

Dla kontenera sekwencyjnego, ograniczona przez koncept `Sequence`:

```
template<Sequence S>
void deleteElementFromContainer(S &seq,
typename S::iterator it){
    seq.erase(it);
}
```

Dla kontenera sekwencyjnego *forward*, ograniczona przez koncept `SequenceForward`:

```
template<SequenceForward S>
void deleteElementFromContainer(S &seq,
typename S::iterator it){
    seq.remove(*it);
}
```



```
}
```

Dla kontenera asocjacyjnego, ograniczona przez koncept `Associative_container`:

```
template<Associative_container S>
void deleteElementFromContainer(S &seq ,
typename S::iterator it){
    seq.erase(it);
}
```

Funkcja usuwa element z kontenera.