Maciej Knapik
28 stycznia 2021

Implementacja grafów oparta na liście sąsiedztwa

URUCHOMIENIE

1. Aby uruchomić program testujący należy z poziomu terminala wywołać kolejno polecenia:
    - make
    - ./main.out

IMPLEMENTACJA

1. Element listy sąsiedztwa

```
template <typename T>
struct graphEdge {
    graphEdge * next;
    T value;
};
```

    - Zdefiniowanie elementu listy sąsiedztwa (lista jednokierunkowa oparta na wskaźnikach) złożonego z wartości (*value*) oraz wskaźnika do kolejnego elementu (*next)*,

2. Atrybuty klasy

```
graphEdge<T> ** A;
bool directed;

int vertexNumber;
int edgeNumber = 0;
```

    - *A* - tablica przechowująca listy sąsiedztwa grafu,
    - *directed* - zmienna określająca, czy graf jest skierowany, czy nie,
    - *vertexNumber* - liczba wierzchołków w grafie,
    - *edgeNumber* - liczba krawędzi w grafie,

3. Konstruktor

```
Graph(int n, bool dir) {
    vertexNumber = n;
    directed = dir;
    A = new graphEdge<T> * [vertexNumber];
}
```

- Przekazanie rozmiaru grafu, czyli ilości wierzchołków w grafie (*n*, a dalej *vertexNumber*), rozmiar grafu jest stały, zmienna jest tylko liczba krawędzi,
- Określenie, czy graf jest skierowany (*dir,* a dalej *directed*),
- Utworzenie tablicy list sąsiedztwa (*A*),

4. Zwracanie informacji czy graf jest skierowany

```
bool is_directed() const { return directed; }
```

- Metoda zwraca informację, czy graf jest skierowany,

5. Zwracanie liczby wierzchołków

```
int getV() { return vertexNumber; }
```

- Metoda zwraca rozmiar grafu, czyli liczbę wierzchołków w grafie,

6. Zwracanie liczby krawędzi

```
int getE() { return edgeNumber; }
```

- Metoda zwraca liczbę krawędzi w grafie,

7. Dodawanie krawędzi

```
bool Graph<T>::addEdge(int v1, int v2)  {

    if(v1 < vertexNumber && v2 < vertexNumber && !hasEdge(v1,v2)) {
        graphEdge<T> *p;

        p = new graphEdge<T>;
        p->value = v2;
        p->next = A[v1];
        A[v1] = p;

        if(!directed) {
            addEdge(v2,v1);
        }

        edgeNumber++;
        return true;
    } else return false;
}
```

- Metoda zwraca fałsz, jeśli jeden z wierzchołków rozpinających krawędź nie należy do grafu, bądź jeśli już posiada daną krawędź,
- Metoda tworzy nowy element listy, ustawia jego wskaźnik na pierwszy element listy, a następnie ustawia początek listy na nowy element,
- Jeśli graf jest nieskierowany, metoda wywołuje samą siebie, aby utworzyć krawędź “w drugą stronę”,
- Metoda zwiększa licznik krawędzi i zwraca prawdę,

8. Usuwanie krawędzi

```
void Graph<T>::removeEdge(int v1, int v2) {
    if(hasEdge(v1, v2)) {
        graphEdge<T> *p, *q;
        p = A[v1];

        if(A[v1]->value == v2) {
            A[v1] = p->next;
            delete p;
        } else {
            while(p->next->value != v2) {
                p = p->next;
            }
            q = p->next;
            p->next = q->next;
            delete q;
        }

        if(!directed) {
            removeEdge(v2, v1);
        }

        edgeNumber--;
    }
}
```

- Metoda sprawdza, czy krawędź do usunięcia istnieje,
- Metoda przekazuje wskaźniki w liście poza element, a następnie go usuwa,

9. Zwracanie informacji, czy graf posiada daną krawędź,

```
bool Graph<T>::hasEdge(int v1, int v2) {
    graphEdge<T> *p;
    p = A[v1];
    while(p) {
        if(p->value == v2)
            return true;
        p = p->next;
    }
    return false;
}
```

- Metoda zwraca prawdę, jeśli dana krawędź istnieje w grafie, bądź fałsz, jeśli nie istnieje,

10. Wyświetlanie grafu,

```
void Graph<T>::display() {
    graphEdge<T> *p;
    for(int i = 0; i < vertexNumber; i++) {
            std::cout << "A[" << i << "] = ";
            p = A[i];
            while(p) {
                std::cout << " " << p->value;
                p = p->next;
            }
            std::cout << std::endl;
    }
}
```

- Metoda wyświetla listę sąsiedztwa

```
A[0] =  1
A[1] =  4 3 2 0
A[2] =  1
A[3] =  1
A[4] =  1
```

11. Czyszczenie grafu,

```
void Graph<T>::clear() {
    graphEdge<T> *p;
    for(int i = 0; i < vertexNumber; i++) {
        p = A[i];
        while(p) {
            removeEdge(i, p->value);
            p = A[i];
        }
    }
}
```

- Metoda usuwa kolejne krawędzie grafu,

12. Przechodzenie grafu w głąb,

```
template<typename T>
void Graph<T>::DFS(int v, bool visited[]) {
    graphEdge<T> *p;
    visited[v] = true;
    std::cout << v << " ";

    for(p = A[v]; p; p = p->next)
        if(!visited[p->value])
            DFS(p->value, visited);
}
```

- Algorytm rekurencyjny DFS,
- Metoda przyjmuje w argumentach aktualny wierzchołek oraz tablicę odwiedzonych wierzchołków,
- Następuje odwiedzenie i wypisanie aktualnego wierzchołka,
- Następuje odwiedzenie nieodwiedzonych sąsiadów,

13. Przechodzenie grafu wszerz

```
void Graph<T>::BFS(int v) {
    graphEdge<T> *p;
    bool *visited = new bool[vertexNumber];
    for(int i = 0; i < vertexNumber; i++) {
        visited[i] = false;
    }

    std::queue<T> queue;

    visited[v] = true;
    queue.push(v);

    while(!queue.empty()) {
        v = queue.front();
        std::cout << v << " ";
        queue.pop();

        for(p = A[v]; p; p = p->next) {
            if(!visited[p->value]) {
                visited[p->value] = true;
                queue.push(p->value);
            }
        }
    }
}
```

- Algorytm rekurencyjny BFS,
- Metoda tworzy kolejkę,
- Aktualny wierzchołek zostaje odwiedzony oraz umieszczony w kolejce,
- Odczytanie wierzchołka z kolejki, usunięcie i wstawienie oraz oznaczenie jako odwiedzonych wszystkich sąsiadów wierzchołka,
- Pętla wykonuje się do momentu wyczyszczenia kolejki, czyli odwiedzenia wszystkich wierzchołków w grafie,