**Державний вищий навчальний заклад  
Ужгородський національний університет  
Факультет інформаційних технологій**

ПРАКТИЧНА РОБОТА № 7  
**Тема:** Алгоритми на Графах. Алгоритм Дейкстри

Виконав студент  
І курсу спеціальності «Інженерія програмного забезпечення»

Романюк Артем

**Ужгород-2025**

**Мета:** навчитися реалізовувати алгоритм Дейкстри для пошуку найкоротшого шляху від однієї вершини графа до інших.

**Питання для самоконтролю**

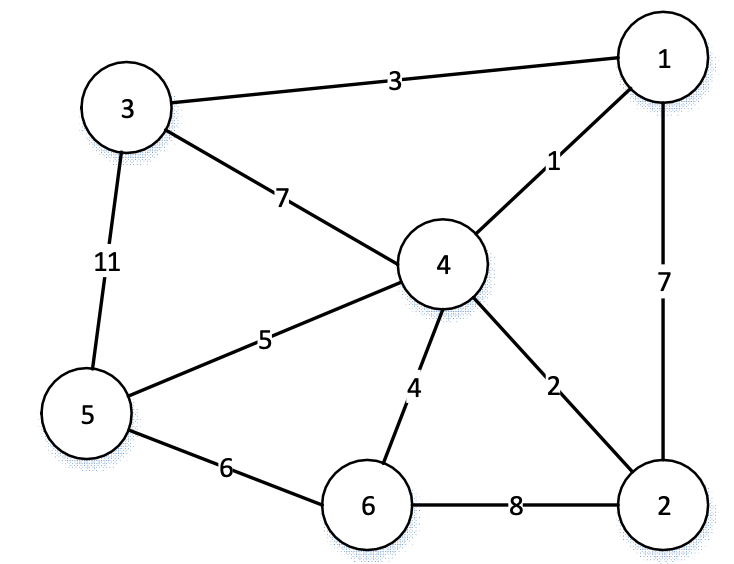
1. Що таке граф?
2. Які є види графів ?
3. Як можуть задаватися графи?
4. Що таке вершина у графі?
5. Що таке ребро у графі?
6. Які алгоритми для пошуку мінімальної відстані у графі вам відомі?

Завдання до роботи:

1. Написати код програми, яка реалізує алгоритм Дейкстри, відповідно до обраного варіанту. Варіант, де вершини виділені повинен виконувати конкретне завдання. Там, де вершини не виділені повинна бути можливість задати вершин для пошуку шляху.
2. Протестувати роботу програми використовуючи різні вхідні дані.
3. Обґрунтувати хід виконання програми.
4. Оформити звіт та завантажити його в системі електронного навчання ДВНЗ «УжНУ» в установлений термін.
5. Підготувати відповіді на контрольні питання.

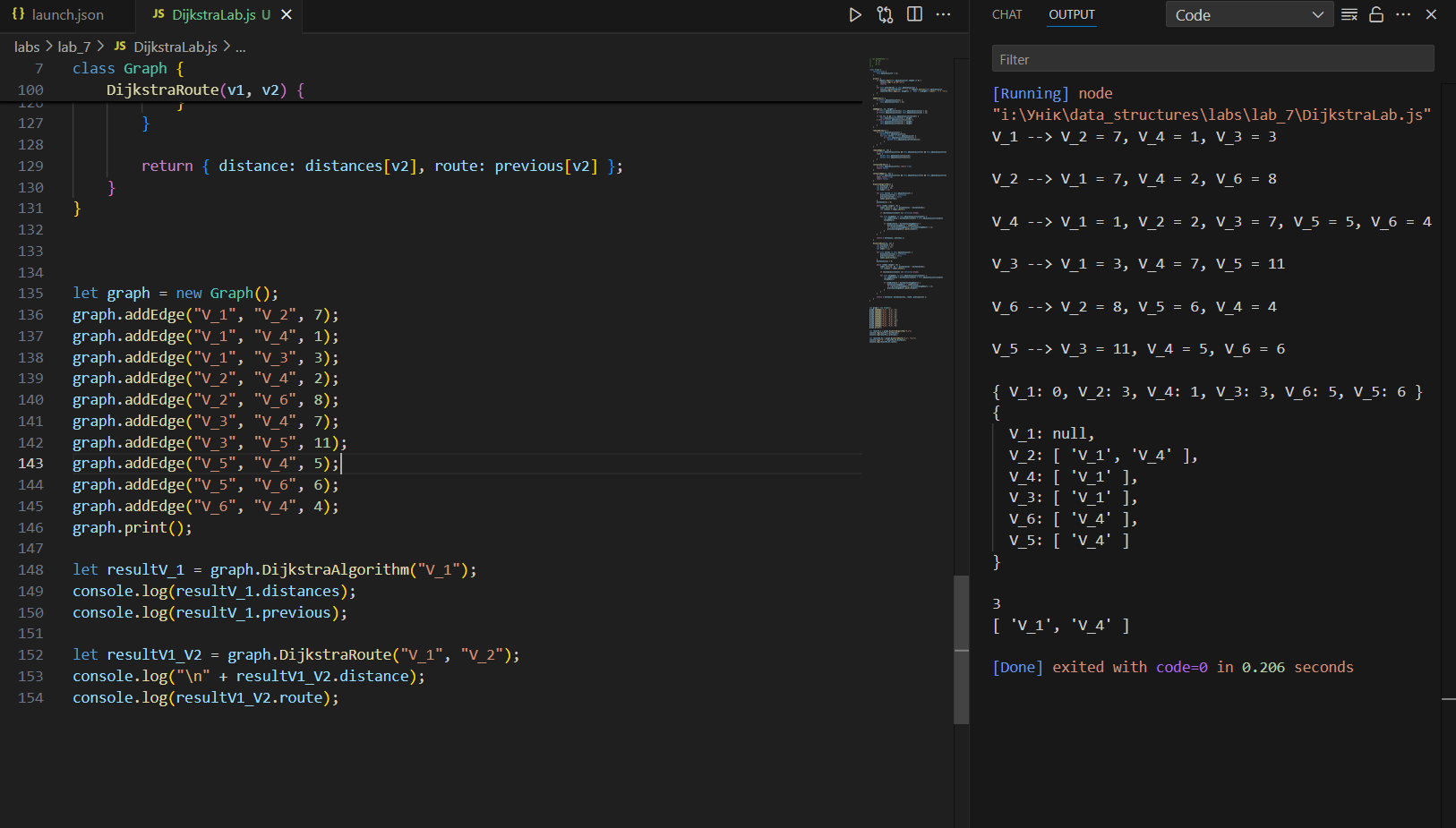
**Хід роботи:**

**Варіант 7**



Реалізація в коді:

| // let testVertex = {  // B: 12,  // C: 3,  // D: 6,  // }  class Graph {  constructor() {  this.adjacencyList = {};  }  print() {  if (Object.keys(this.adjacencyList).length <= 0) {  console.log("Граф пустий");  return;  }  for (let outerVertex in this.adjacencyList) {  console.log(outerVertex + " --> " + Object.entries(this.adjacencyList[outerVertex]).map(([v, weight]) => `${v} = ${weight}`).join(", ") + "\n");  }  }  addVertex(v) {  if (!this.adjacencyList[v]) {  this.adjacencyList[v] = {};  }  }  addEdge(v1, v2, weight) {  if (!this.adjacencyList[v1]) this.adjacencyList[v1] = {};  if (!this.adjacencyList[v2]) this.adjacencyList[v2] = {};  if (v1 === v2 && !this.adjacencyList[v1][v2]) {  this.adjacencyList[v1][v2] = weight;  } else if (!this.adjacencyList[v1][v2]) {  this.adjacencyList[v1][v2] = weight;  this.adjacencyList[v2][v1] = weight;  }  }  removeVertex(v) {  if (this.adjacencyList[v]) {  delete this.adjacencyList[v];  for (let vertex in this.adjacencyList) {  if (this.adjacencyList[vertex][v]) {  delete this.adjacencyList[vertex][v];  }  }  }  }  removeEdge(v1, v2) {  if (this.adjacencyList[v1] && this.adjacencyList[v2] && this.adjacencyList[v1][v2]) {  delete this.adjacencyList[v1][v2];  delete this.adjacencyList[v2][v1];  }  }  containsVertex(v) {  if (this.adjacencyList[v]) return true;  return false;  }  containsEdge(v1, v2) {  if (this.adjacencyList[v1] && this.adjacencyList[v2] && this.adjacencyList[v1][v2]) return true;  return false;  }  DijkstraAlgorithm(v) {  let distances = {};  let previous = {};  let nodes = [];  for (let vertex in this.adjacencyList) {  distances[vertex] = Infinity;  previous[vertex] = null;  nodes.push(vertex);  }  distances[v] = 0;  while (nodes.length > 0) {  nodes.sort((a, b) => distances[a] - distances[b]);  let closest = nodes.shift();  if (distances[closest] === Infinity) break;  for (let neighbour in this.adjacencyList[closest]) {  let newDistance = distances[closest] + this.adjacencyList[closest][neighbour];  if (newDistance < distances[neighbour]) {  distances[neighbour] = newDistance;  if (!previous[neighbour]) previous[neighbour] = [];  previous[neighbour].push(closest);  }  }  }  return { distances, previous };  }  DijkstraRoute(v1, v2) {  let distances = {};  let previous = {};  let nodes = [];  for (let vertex in this.adjacencyList) {  distances[vertex] = Infinity;  previous[vertex] = null;  nodes.push(vertex);  }  distances[v1] = 0;  while (nodes.length > 0) {  nodes.sort((a, b) => distances[a] - distances[b]);  let closest = nodes.shift();  if (distances[closest] === Infinity) break;  for (let neighbour in this.adjacencyList[closest]) {  let newDistance = distances[closest] + this.adjacencyList[closest][neighbour];  if (newDistance < distances[neighbour]) {  distances[neighbour] = newDistance;  if (!previous[neighbour]) previous[neighbour] = [];  previous[neighbour].push(closest);  }  }  }  return { distance: distances[v2], route: previous[v2] };  }  }  let graph = new Graph();  graph.addEdge("V\_1", "V\_2", 7);  graph.addEdge("V\_1", "V\_4", 1);  graph.addEdge("V\_1", "V\_3", 3);  graph.addEdge("V\_2", "V\_4", 2);  graph.addEdge("V\_2", "V\_6", 8);  graph.addEdge("V\_3", "V\_4", 7);  graph.addEdge("V\_3", "V\_5", 11);  graph.addEdge("V\_5", "V\_4", 5);  graph.addEdge("V\_5", "V\_6", 6);  graph.addEdge("V\_6", "V\_4", 4);  graph.print();  let resultV\_1 = graph.DijkstraAlgorithm("V\_1");  console.log(resultV\_1.distances);  console.log(resultV\_1.previous);  let resultV1\_V2 = graph.DijkstraRoute("V\_1", "V\_2");  console.log("\n" + resultV1\_V2.distance);  console.log(resultV1\_V2.route); |
| --- |



**Висновок:**

У ході виконання практичної роботи було реалізовано алгоритм Дейкстри на мові програмування JavaScript для знаходження найкоротшого шляху у неорієнтованому зваженому графі. Програма дозволяє як знаходити відстані від однієї вершини до всіх інших, так і визначати найкоротший шлях між двома конкретними вершинами. Результати роботи алгоритму були протестовані на прикладі графа з шістьма вершинами, і виведені дані підтвердили правильність реалізації. Таким чином, мету роботи досягнуто, а набуті знання можуть бути застосовані при вирішенні задач оптимізації маршрутів у мережах.