**Міністерство освіти і науки України**

**Сумський державний університет**

**Кафедра комп'ютерних наук**

**ЗВІТ**

**з дисципліни "Алгоритми і структури даних"**

**Виконав: Штогрін Вячеслав**

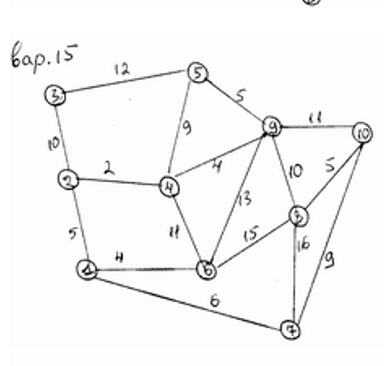
**Перевірив: Шаповалов С.П.**

**Група: ІН-11/2**

Лабораторна робота № 11-12

**«АЛГОРИТМИ Прима та Крускала»**

Теоретичні відомості



**Основне дерево (кістякове дерево, каркас)** є ациклічним підграфом зв'язного неорієнтованого графа, до складу якого входять всі вершини даного графа. Якщо заданим графом є граф з вагами ребер, то можна ставити завдання про пошук остового дерева мінімальної ваги. Така задача в CS має абревіатуру MST (Minimum spanning tree). Основними алгоритмами розв’язання проблеми MST є алгоритми Прима й Крускала (Краскала), що притримуються жадібної стратегії.

*Алгоритми Прима та Крускала мають узагальнений псевдокод*

Generic MST(G, w)

1 A**←**Ø

2 while A не є мінімальним остовим деревом

3 do знайти для А безпечне ребро (u, v)

4 A ∪ {(u, v)} / додаємо безпечне ребро до остова

5 return A

Обидва алгоритми притримуються загальної парадигми жадібної стратегії, але кожний з них використовує своє правило для визначення безпечного ребра для додавання до остового дерева. В алгоритмі Крускала множина А являє собою ліс й безпечне ребро об’єднує дві різні компоненти в лісі (два дерева в одне), а в алгоритмі Прима множина А є єдиним деревом, тому безпечне ребро з’єднує це дерево з вершиною зовні дерева.

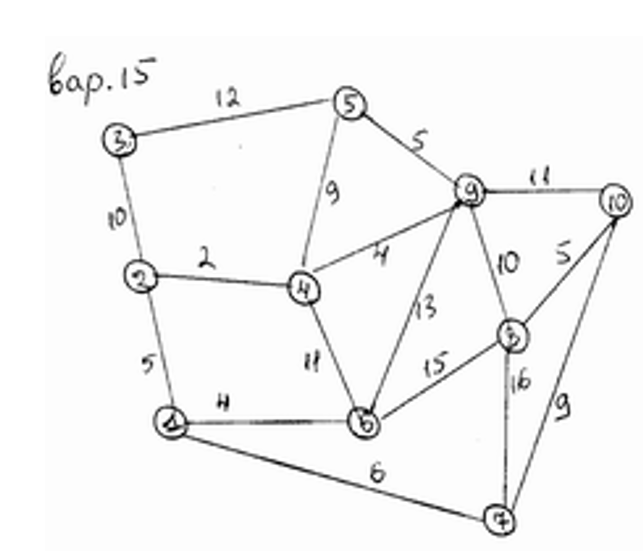
Завдання ЛАБОРОТОРНОЇ роботи

1. Створити комп’ютерну реалізацію алгоритмів Прима та Крускала для графа за варіан-том завдання. Написати програму (функцію main), що містить опис заданих графів та процедури знаходження остових дерев за алгоритмами.

2. Порівняти виконання алгоритмів за часом та кількістю операцій при виконанні.

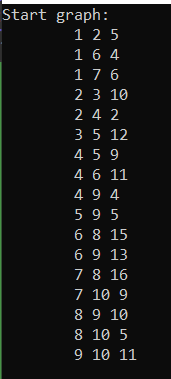
3. Вивести в консоль результати.

4. Заповнити порівняльну таблицю.

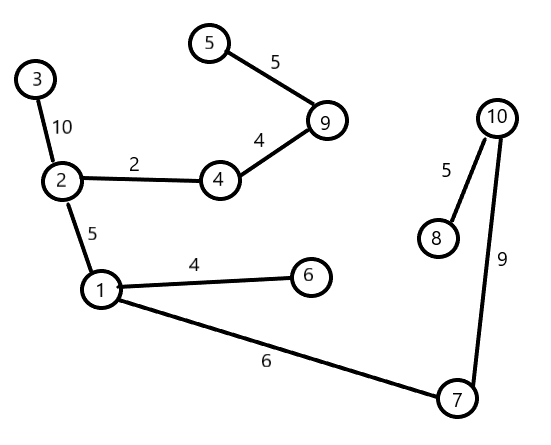
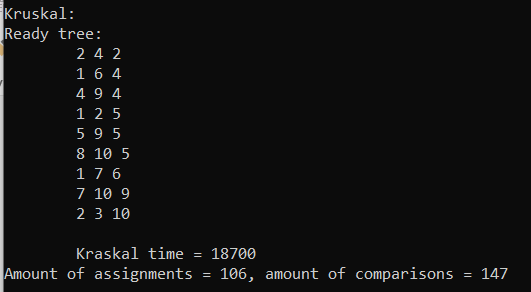


Програмний код

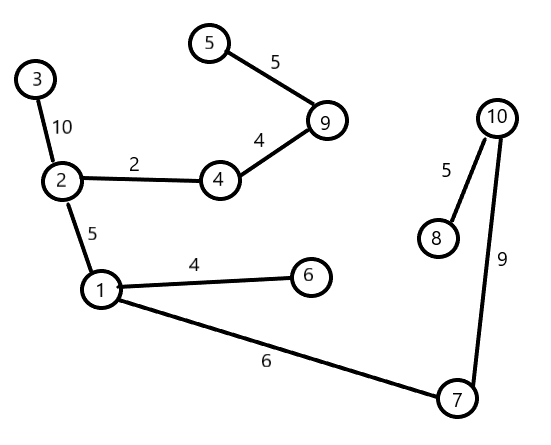
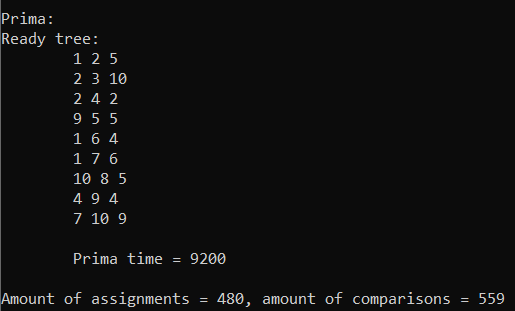
|  |
| --- |
| #include <iostream>  #include <vector>  #include <algorithm>  #include <chrono>  using namespace std;  class Edge {  public:  int m\_numberOfVertex\_1;  int m\_numberOfVertex\_2;  float m\_weight;  public:  Edge(int numberOfVertex\_1, int numberOfVertex\_2, float weight) :  m\_numberOfVertex\_1(numberOfVertex\_1),  m\_numberOfVertex\_2(numberOfVertex\_2),  m\_weight(weight) {}  friend bool operator < (Edge& first, Edge& second) {  if (first.m\_weight < second.m\_weight) {  return true;  }  return false;  }  };  class Kruskal {  protected:  vector<Edge> m\_startGraph;  vector<Edge> m\_readyTree;  int\* m\_collisions;  public:  pair<int, int> amount\_op{ 0,0 };  protected:  void initialize() {  sort(m\_startGraph.begin(), m\_startGraph.end());  m\_collisions = new int[m\_startGraph.size()];  amount\_op.first += 2;  amount\_op.second += 1;  for (int i = 0; i < m\_startGraph.size(); i++) {  amount\_op.second += 1;  amount\_op.first += 2;  m\_collisions[i] = i;  }  }  int find\_set(int position) {  amount\_op.second += 1;  if (position == m\_collisions[position]) {  return position;  }  else {  return find\_set(m\_collisions[position]);  }  }  public:  Kruskal() {}  void addEdge(int numberOfVertex\_1, int numberOfVertex\_2, float weight) {  m\_startGraph.push\_back(Edge(numberOfVertex\_1, numberOfVertex\_2, weight));  }  void calculateTree() {  int collisionNumber\_1, collisionNumber\_2;  initialize();  amount\_op.first += 1;  amount\_op.second += 1;  for (int i = 0; i < m\_startGraph.size(); i++) {  amount\_op.first += 3;  amount\_op.second += 2;  collisionNumber\_1 = find\_set(m\_startGraph[i].m\_numberOfVertex\_1);  collisionNumber\_2 = find\_set(m\_startGraph[i].m\_numberOfVertex\_2);  if (collisionNumber\_1 != collisionNumber\_2) {  amount\_op.first += 2;  amount\_op.second += 1;  m\_readyTree.push\_back(m\_startGraph[i]);  m\_collisions[collisionNumber\_1] = m\_collisions[collisionNumber\_2];  }  }  }  void printStartGraph() {  cout << "Start graph: " << endl;  for (int i = 0; i < m\_startGraph.size(); i++) {  cout << "\t" << m\_startGraph[i].m\_numberOfVertex\_1 << " " << m\_startGraph[i].m\_numberOfVertex\_2 << " " << m\_startGraph[i].m\_weight << endl;  }  cout << endl;  }  void printReadyTree() {  cout << "Ready tree: " << endl;  for (int i = 0; i < m\_readyTree.size(); i++) {  cout << "\t" << m\_readyTree[i].m\_numberOfVertex\_1 << " " << m\_readyTree[i].m\_numberOfVertex\_2 << " " << m\_readyTree[i].m\_weight << endl;  }  cout << endl;  }  vector<Edge> getStartGraph() {  return m\_startGraph;  }  };  class Prima {  protected:  vector<Edge> m\_startGraph;  vector<Edge> m\_readyTree;  int m\_sizeOfMAtrix;  int\*\* m\_matrix;  public:  pair<int, int> amount\_op{ 0,0 };  protected:  int findSizeOfMatrix() {  int max\_1 = m\_startGraph[0].m\_numberOfVertex\_1;  int max\_2 = m\_startGraph[0].m\_numberOfVertex\_2;  for (int i = 1; i < m\_startGraph.size(); i++) {  if (m\_startGraph[i].m\_numberOfVertex\_1 > max\_1) {  max\_1 = m\_startGraph[i].m\_numberOfVertex\_1;  }  if (m\_startGraph[i].m\_numberOfVertex\_2 > max\_2) {  max\_2 = m\_startGraph[i].m\_numberOfVertex\_2;  }  }  return max\_1 > max\_2 ? max\_1 : max\_2;  }  void convertGraphIntoMatrix() {  m\_sizeOfMAtrix = findSizeOfMatrix();  m\_matrix = new int\* [m\_sizeOfMAtrix];  for (int i = 0; i < m\_sizeOfMAtrix; i++) {  m\_matrix[i] = new int[m\_sizeOfMAtrix];  for (int j = 0; j < m\_sizeOfMAtrix; j++) {  m\_matrix[i][j] = 0;  }  }  for (int i = 0; i < m\_startGraph.size(); i++) {  m\_matrix[m\_startGraph[i].m\_numberOfVertex\_1 - 1][m\_startGraph[i].m\_numberOfVertex\_2 - 1] = m\_startGraph[i].m\_weight;  m\_matrix[m\_startGraph[i].m\_numberOfVertex\_2 - 1][m\_startGraph[i].m\_numberOfVertex\_1 - 1] = m\_startGraph[i].m\_weight;  }  }  int minKey(int\* key, bool\* mstSet) {  amount\_op.first += 1;  int min = INT\_MAX, min\_index;  for (int v = 0; v < m\_sizeOfMAtrix; v++) {  amount\_op.first += 2;  amount\_op.second += 3;  if (mstSet[v] == false && key[v] < min) {  min = key[v], min\_index = v;  }  }  return min\_index;  }  public:  Prima() {}  void setStartGraph(vector<Edge> startGraph) {  m\_startGraph = startGraph;  convertGraphIntoMatrix();  }  void printStartGraph() {  cout << "Start graph: " << endl;  for (int i = 0; i < m\_startGraph.size(); i++) {  cout << "\t" << m\_startGraph[i].m\_numberOfVertex\_1 << " " << m\_startGraph[i].m\_numberOfVertex\_2 << " " << m\_startGraph[i].m\_weight << endl;  }  cout << endl;  }  void printReadyTree() {  cout << "Ready tree: " << endl;  for (int i = 0; i < m\_readyTree.size(); i++) {  cout << "\t" << m\_readyTree[i].m\_numberOfVertex\_1 << " " << m\_readyTree[i].m\_numberOfVertex\_2 << " " << m\_readyTree[i].m\_weight << endl;  }  cout << endl;  }  void calculateTree() {  amount\_op.first += 5;  int\* parent = new int[m\_sizeOfMAtrix];  int\* key = new int[m\_sizeOfMAtrix];  bool\* mstSet = new bool[m\_sizeOfMAtrix];  for (int i = 0; i < m\_sizeOfMAtrix; i++) {  key[i] = INT\_MAX;  mstSet[i] = false;  amount\_op.first += 4;  amount\_op.second += 1;  }  key[0] = 0;  parent[0] = -1;  for (int count = 0; count < m\_sizeOfMAtrix - 1; count++) {  amount\_op.first += 4;  amount\_op.second += 1;  int u = minKey(key, mstSet);  mstSet[u] = true;  for (int v = 0; v < m\_sizeOfMAtrix; v++) {  amount\_op.first += 2;  amount\_op.second += 3;  if (m\_matrix[u][v] and mstSet[v] == false and m\_matrix[u][v] < key[v]) {  amount\_op.first += 2;  parent[v] = u;  key[v] = m\_matrix[u][v];  }  }  }  for (int i = 1; i < m\_sizeOfMAtrix; i++) {  m\_readyTree.push\_back(Edge(parent[i] + 1, i + 1, m\_matrix[i][parent[i]]));  }  }  };  int main() {  Kruskal kruskal;  Prima prima;  kruskal.addEdge(1, 2, 5);  kruskal.addEdge(1, 6, 4);  kruskal.addEdge(1, 7, 6);  kruskal.addEdge(2, 3, 10);  kruskal.addEdge(2, 4, 2);  kruskal.addEdge(3, 5, 12);  kruskal.addEdge(4, 5, 9);  kruskal.addEdge(4, 6, 11);  kruskal.addEdge(4, 9, 4);  kruskal.addEdge(5, 9, 5);  kruskal.addEdge(6, 8, 15);  kruskal.addEdge(6, 9, 13);  kruskal.addEdge(7, 8, 16);  kruskal.addEdge(7, 10, 9);  kruskal.addEdge(8, 9, 10);  kruskal.addEdge(8, 10, 5);  kruskal.addEdge(9, 10, 11);  cout << "Kruskal:" << endl;  kruskal.printStartGraph();  auto start\_time = std::chrono::high\_resolution\_clock::now();  kruskal.calculateTree();  auto finish = std::chrono::high\_resolution\_clock::now();  kruskal.printReadyTree();  //cout << "\t-Number of assignment operations = " << amount\_oper.first << ", number of comparisons operations = " << amount\_oper.second << endl;  cout << "\tKraskal time = " << std::chrono::duration\_cast<std::chrono::nanoseconds>(finish - start\_time).count() << endl;  cout << "Amount of assignments = " << kruskal.amount\_op.first << ", amount of comparisons = " << kruskal.amount\_op.second << endl << endl;;  cout << "Prima:" << endl;  prima.setStartGraph(kruskal.getStartGraph());  start\_time = std::chrono::high\_resolution\_clock::now();  prima.calculateTree();  finish = std::chrono::high\_resolution\_clock::now();  prima.printReadyTree();  cout << "\tPrima time = " << std::chrono::duration\_cast<std::chrono::nanoseconds>(finish - start\_time).count() << endl << endl;  cout << "Amount of assignments = " << prima.amount\_op.first << ", amount of comparisons = " << prima.amount\_op.second << endl;  system("pause");  return 0;  } |



**Алгоритм Крускала**



**Алгоритм Прима**



Порівняльна таблиця

|  |  |  |  |
| --- | --- | --- | --- |
|  | Час | Призначень | Порівняння |
| Kruskal | 18700 | 106 | 147 |
| Prima | 9200 | 480 | 559 |