Харківський національний університет імені В. Н. Каразіна
Факультет комп'ютерних наук
Кафедра штучного інтелекту та програмного забезпечення

ЗВІТ
З ПРАКТИЧНОЇ РОБОТИ №11

дисципліна: «Алгоритми та структур и даних»

Виконала: студентка групи КС-22
Узенкова Дар’я
Перевірив: Олешко Олег

Харків
2024

1. Реалізувати алгоритм Прима знаходження мінімального кістякового дерева. Для представлення графа використовувати матрицю суміжності, для черги з пріоритетами – масив.

Виконати порівняльний аналіз ефективності алгоритму для розріджених та щільних графів.

2. Реалізувати алгоритм Крускала знаходження мінімального кістякового дерева.

Виконати порівняльний аналіз ефективності з алгоритмом Пріма на графах для попереднього алгоритму (для розріджених та щільних графів).

Результати виконання завдання наведено:

1. У лістингу 1, 2 – вихідні коди програми алгоритмів Прима та Крускала.
2. У малюнках 1, 2, 3, 4 – результати виконання програми.

Алгоритм Прима

```c
#include <stdio.h>
#include <stdbool.h>
#include <time.h>
#include <limits.h>

#define INF 9999999

// Функція для знаходження індексу мінімального елемента в масиві key, який ще не включений в MST
int minKey(int key[], bool selected[], int V) {
    int min = INF, min_index;

    for (int v = 1; v <= V; v++) {  // починаємо з 1
        if (!selected[v] && key[v] < min) {
            min = key[v];
            min_index = v;
        }
    }
    return min_index;
}

// Функція для виведення MST, представленого масивом parent[]
void printMST(int parent[], int graph[][101], int V) {
    printf("Ребро \tВага\n");
    for (int i = 2; i <= V; i++) {
        printf("%d - %d \t%d \n", parent[i], i, graph[i][parent[i]]);
    }
}
```

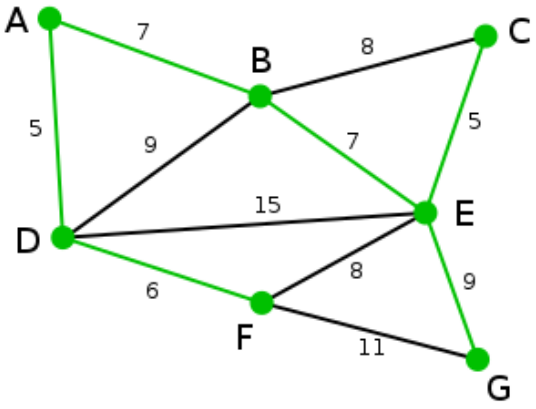
```c
// Основна функція для реалізації алгоритму Прима з використанням масиву як черги з пріоритетом
void primMST(int graph[][101], int V) {
    int parent[V + 1];   // Масив для зберігання побудованого MST
    int key[V + 1];      // Масив для відстеження мінімальної ваги для кожної вершини
    bool selected[V + 1]; // Масив для відстеження включення вершини в MST

    // Ініціалізація ключів як нескінченних і selected[] як false
    for (int i = 1; i <= V; i++) {
        key[i] = INF;
        selected[i] = false;
    }

    // Починаємо з першої вершини
    key[1] = 0;
    parent[1] = -1; // Перший вузол завжди корінь MST

    for (int count = 1; count < V; count++) {
        int u = minKey(key, selected, V); // Обираємо вершину з мінімальною вагою
        selected[u] = true; // Включаємо обрану вершину в MST

        // Оновлюємо ваги суміжних вершин, які ще не включені в MST
        for (int v = 1; v <= V; v++) {
            if (graph[u][v] && !selected[v] && graph[u][v] < key[v]) {
                parent[v] = u;
                key[v] = graph[u][v];
            }
        }
    }

    // Виводимо побудоване MST
    printMST(parent, graph, V);
}

int main() {
    int V, E;

    printf("Введіть кількість вершин: ");
    scanf("%d", &V);

    int graph[101][101];

    // Ініціалізуємо матрицю суміжності нулями
    for (int i = 1; i <= V; i++) {
        for (int j = 1; j <= V; j++) {
            graph[i][j] = 0;
        }
    }

    printf("Введіть кількість ребер: ");
```

```
    scanf("%d", &E);

    printf("Введіть ребра у форматі (початок кінець вага):\n");
    for (int i = 0; i < E; i++) {
        int u, v, weight;
        scanf("%d %d %d", &u, &v, &weight);
        graph[u][v] = weight;
        graph[v][u] = weight; // Для неорієнтованого графу
    }

    printf("\n");

     clock_t start = clock();

    primMST(graph, V);

    clock_t end = clock();

    double time_taken = ((double)(end - start) / CLOCKS_PER_SEC) * 1e6;
    printf("\nЧас виконання алгоритму Прима: %.0f мкс\n", time_taken);

    return 0;
}
```

Лістинг 1 – вихідний код програми знаходження мінімального кістякового дерева алгоритмом Прима

```
Введіть кількість вершин: 7
Введіть кількість ребер: 11
Введіть ребра у форматі (початок кінець вага):
1 2 7
2 3 8
3 5 5
5 2 7
1 4 5
4 2 9
4 5 15
4 6 6
6 5 8
5 7 9
6 7 11

Ребро   Вага
1 - 2   7
5 - 3   5
1 - 4   5
2 - 5   7
4 - 6   6
5 - 7   9

Час виконання алгоритму Прима: 4000 мкс
```

Малюнок 1 – результат виконання програми для розрідженого графа



Розріджений граф (A -1, B – 2, C – 3…)

```
Введіть кількість вершин: 7
Введіть кількість ребер: 21
Введіть ребра у форматі (початок кінець вага):
1 2 3
1 3 10
1 4 5
1 5 8
1 6 7
1 7 6
2 3 12
2 4 9
2 5 7
2 6 4
2 7 10
3 4 11
3 5 6
3 6 8
3 7 9
4 5 7
4 6 3
4 7 2
5 6 4
5 7 5
6 7 1

Ребро   Вага
1 - 2   3
5 - 3   6
7 - 4   2
6 - 5   4
2 - 6   4
6 - 7   1

Час виконання алгоритму Прима: 5000 мкс
```

Малюнок 2 – результат виконання програми для щільного графа

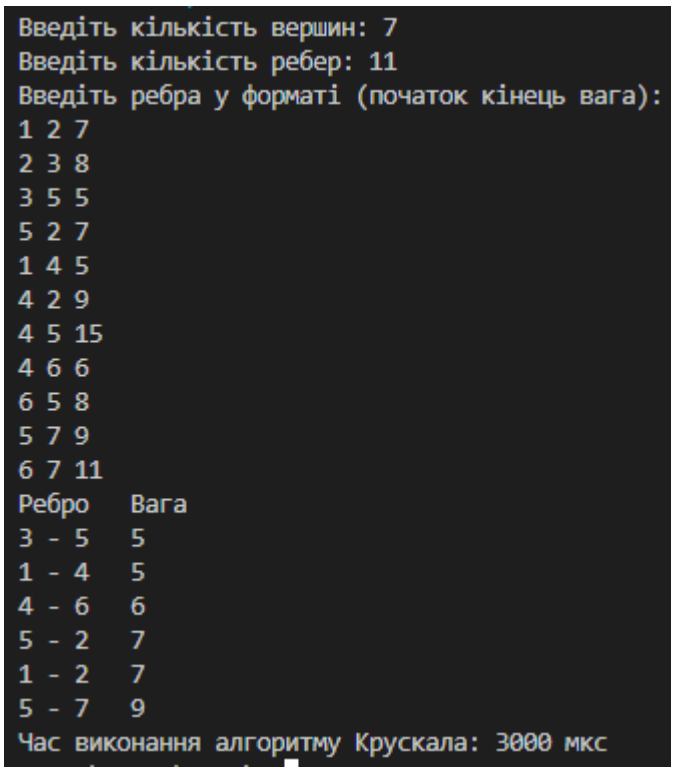
Алгоритм Крускала

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// Структура для представлення ребра графа
struct Edge {
    int start, end, weight;
};

// Структура для представлення графа
struct Graph {
    int V, E; // Кількість вершин і ребер
    struct Edge* edges; // Масив ребер
};

// Структура для представлення підмножини (для алгоритму об'єднання)
struct Subset {
    int parent;
    int rank;
};

// Функція для створення графа
struct Graph* createGraph(int V, int E) {
    struct Graph* graph = (struct Graph*) malloc(sizeof(struct Graph));
    graph->V = V;
    graph->E = E;
    graph->edges = (struct Edge*) malloc(E * sizeof(struct Edge));
    return graph;
}

// Функція для знаходження підмножини, до якої належить елемент i
int find(struct Subset subsets[], int i) {
    if (subsets[i].parent != i) {
        subsets[i].parent = find(subsets, subsets[i].parent); // Шляхова
компресія
    }
    return subsets[i].parent;
}

// Функція для об'єднання двох підмножин
void unionSubsets(struct Subset subsets[], int x, int y) {
    int xroot = find(subsets, x);
    int yroot = find(subsets, y);

    // Приєднуємо дерево меншого рангу до дерева більшого рангу
    if (subsets[xroot].rank < subsets[yroot].rank) {
        subsets[xroot].parent = yroot;
    } else if (subsets[xroot].rank > subsets[yroot].rank) {
        subsets[yroot].parent = xroot;
```

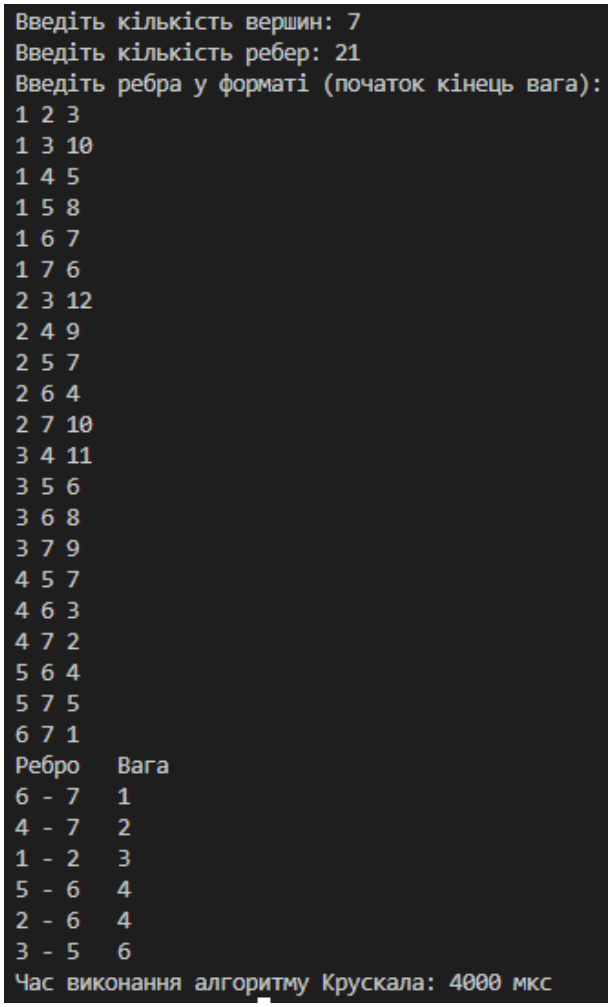
```
    } else {
        subsets[yroot].parent = xroot;
        subsets[xroot].rank++;
    }
}

// Функція для сортування ребер за вагою
int compareEdges(const void* a, const void* b) {
    struct Edge* edgeA = (struct Edge*)a;
    struct Edge* edgeB = (struct Edge*)b;
    return edgeA->weight - edgeB->weight;
}

// Функція реалізації алгоритму Крускала
void kruskalMST(struct Graph* graph) {
    int V = graph->V;
    struct Edge result[V]; // Масив для зберігання MST
    int e = 0; // Поточний індекс результатного масиву
    int i = 0; // Поточний індекс для ітерації ребер

    // Сортуємо всі ребра за вагою
    qsort(graph->edges, graph->E, sizeof(graph->edges[0]), compareEdges);

    // Створюємо підмножини для кожної вершини
    struct Subset* subsets = (struct Subset*) malloc(V * sizeof(struct Subset));
    for (int v = 0; v < V; v++) {
        subsets[v].parent = v;
        subsets[v].rank = 0;
    }

    // Проходимо по ребрах і додаємо їх у MST
    while (e < V - 1 && i < graph->E) {
        struct Edge nextEdge = graph->edges[i++];
        int x = find(subsets, nextEdge.start);
        int y = find(subsets, nextEdge.end);

        // Якщо додавання ребра не створює цикл, додаємо його до MST
        if (x != y) {
            result[e++] = nextEdge;
            unionSubsets(subsets, x, y);
        }
    }

    // Виведення MST
    printf("Ребро \tВага\n");
    for (i = 0; i < e; i++) {
        printf("%d - %d \t%d\n", result[i].start + 1, result[i].end + 1,
result[i].weight);
    }

    free(subsets);
```

```
}

int main() {
    int V, E;

    printf("Введіть кількість вершин: ");
    scanf("%d", &V);
    printf("Введіть кількість ребер: ");
    scanf("%d", &E);

    struct Graph* graph = createGraph(V, E);

    printf("Введіть ребра у форматі (початок кінець вага):\n");
    for (int i = 0; i < E; i++) {
        scanf("%d %d %d", &graph->edges[i].start, &graph->edges[i].end, &graph-
>edges[i].weight);
        graph->edges[i].start--; // Перетворення до індексації з 0
        graph->edges[i].end--;
    }

    clock_t start = clock();

    kruskalMST(graph);

    clock_t end = clock();

    double time_taken = ((double)(end - start) / CLOCKS_PER_SEC) * 1e6;
    printf("Час виконання алгоритму Крускала: %.0f мкс\n", time_taken);

    free(graph->edges);
    free(graph);
    return 0;
}
```

Лістинг 2 – вихідний код програми знаходження мінімального кістякового дерева алгоритмом Крускала



Малюнок 3 – результат виконання програми для розрідженого графа



Малюнок 2 – результат виконання програми для щільного графа

Теоретична складність алгоритмів:

1. **Алгоритм Прима**:

Для графа, представленого матрицею суміжності, з використанням масиву:

Складність пошуку мінімального ребра для кожної вершини - O(V^2), де V - кількість вершин.

- Для розріджених графів (мало ребер): менш ефективний через квадратичну складність, незалежно від кількості ребер.
- Для щільних графів (багато ребер): працює добре, оскільки перевірка кожної пари вершин усе одно неминуча.

2. **Алгоритм Крускала**:

Складність сортування ребер — O(E logE), де E — кількість ребер.

Перевірка циклів за допомогою "системи непересічних множин" - O(E·α(V)), де α — обмежена функція від розміру множин.

- Для розріджених графів: ефективний, оскільки кількість ребер мала.
- Для щільних графів: працює трохи гірше через більший обсяг сортування ребер, але все одно може бути кращим, якщо E logE менше, ніж V^2.

Висновок: В обох випадках алгоритм Крускала демонструє переваги, оскільки сортування ребер менш ресурсомістке, ніж перевірка кожної вершини та ребра в алгоритмі Прима. Проте алгоритм Прима може бути корисним для дуже щільних графів, де кількість ребер наближається до V^2. У таких випадках його часова складність $O(V^2)$ стає порівняно ефективною. Водночас у Крускала сортування великої кількості ребер (O(ElogE)) може стати слабким місцем.