Санкт-Петербургский Политехнический университет Петра Великого
Институт машиностроения, материалов и транспорта
Кафедра «Мехатроника и роботостроение (при ЦНИИ РТК)»

Курсовая работа

Дисциплина: объектно-ориентированное программирование

Тема: *Kruskal's algorithm*

Работу выполнил(а): Николаева Э. С.
студент группы 3331506/80401

Преподаватель: Кузнецова Е. М.

«__»__________ 2021 г.

Санкт-Петербург

2021 г

Оглавление

1 Введение

Алгоритм Краскала находит минимальный остовный подграф неориентированного взвешенного по ребрам графа.

Если граф связан, алгоритм находит минимальное остовное дерево. Минимальное остовное дерево связного графа — это подмножество ребер, которое формирует дерево, включающее каждую вершину, где сумма весов всех ребер в дереве минимальна. Для несвязного графа минимальный остовной подграф состоит из минимального остовного дерева для каждого связного компонента.

Это жадный алгоритм в теории графов, так как на каждом шаге он добавляет следующее ребро с наименьшим весом, которое не образует цикл, к минимальному остовному лесу. То есть принимает локально оптимальное решение на каждом этапе.

<u>Свойства минимального остова:</u>

- Минимальный остов уникален, если веса всех рёбер различны. В противном случае может существовать несколько минимальных остовов.
- Минимальный остов является остовом с минимальным произведением весов рёбер.
- Минимальный остов является остовом с минимальным весом самого тяжелого ребра.
- Остов максимального веса ищется аналогично остову минимального веса, достаточно поменять знаки всех рёбер на противоположные и выполнить любой из алгоритм минимального остова.
- Минимальное остовное дерево имеет (V - 1) ребер, где V - количество вершин в данном графе.

Области применения:

- → Разработка сетей. Соединении n городов в единую телефонную сеть с минимальной суммарной стоимостью соединений.
- → Производство печатных плат. По аналогии с сетью: мы хотим соединить n контактов проводами с минимальной суммарной стоимостью. (Здесь стоит отметить, что задача о минимальном остовном дереве является упрощением реальности. В самом деле, если соединяемые контакты находятся в вершинах единичного квадрата, разрешается соединять любые его вершины, и вес соединения равен его длине, то минимальное покрывающее дерево будет состоять из трех сторон квадрата. Между тем все его четыре вершины можно электрически соединить двумя пересекающимися диагоналями, суммарная длина которых будет равна $2\sqrt{2}$, что меньше 3 в первом случае).
- → Минимальное остовное дерево может использоваться для визуализации многоаспектных, многомерных данных, например, для отображения их взаимосвязи.
- → Наука,в частности биология, используют многомерные данные для группировки объектов, растений, животных. Минимальное остовное дерево позволяет разбивать их на взаимосвязанные классы, четко отслеживая близкие по строению и характеристикам группы.

2 Описание алгоритма

Шаги для поиска минимального остовного дерева с использованием алгоритма Крускала:

1. Отсортировать все ребра в порядке неубывания их веса.
2. Выбрать самый маленький край. Проверить, образует ли он цикл с уже сформированным остовным деревом. Если цикл не образуется, включить это ребро. В противном случае не включать его.
3. Повторять шаг № 2, пока в остовном дереве не будет (V-1) ребер.

В шаге 2 используется алгоритм Union-Find для обнаружения циклов. Для каждого ребра сделаем подмножества, используя обе вершины ребра. Если обе вершины находятся в одном подмножестве, цикл найден. Реализация Union-Find в худшем случае занимает O(n) времени. Время можно улучшить до O (Log(n)) в худшем случае с помощью объединения по рангу. Идея состоит в том, чтобы всегда прикреплять дерево меньшей глубины под корнем более глубокого дерева.

Второе улучшение - сжатие пути (сгладить дерево при вызове find). Когда find вызывается для элемента x, возвращается корень дерева. Операция find проходит вверх от x, чтобы найти корень. Идея сжатия пути в том, чтобы сделать найденный корень родительским для x, чтобы не приходилось снова проходить все промежуточные узлы. Если x является корнем поддерева, то путь (к корню) от всех узлов под x также сжимается. Временная сложность каждой операции становится даже меньше, чем O(Log(n)).

3 Оценка скорости

Частично об оценке скорости было сказано в пункте «Описание алгоритма».

Для графа с E ребрами и V вершинами можно показать, что алгоритм Крускала работает за время O(E•logE) или O(E•logV). Сортировка ребер занимает O(E•LogE) времени. После сортировки мы перебираем все ребра и применяем алгоритм Union-Find. Операции поиска и объединения могут занять самое большее O(LogV) времени. Таким образом, общая сложность составляет время O (E • LogE + E • LogV). Значение E может быть не более $O(V^2)$, поэтому O(LogV) равно O(LogE). Следовательно, общая временная сложность составляет O(E • logE) или O(E • logV).

4 Анализ алгоритма

Для анализа времени работы алгоритма в зависимости от входных данных использована библиотека <chrono>. Более подробно измерения проводили на участке 0-100 вершин, далее точность уменьшается. Графики полученных измерений представлены на Рисунке 1 и 2.

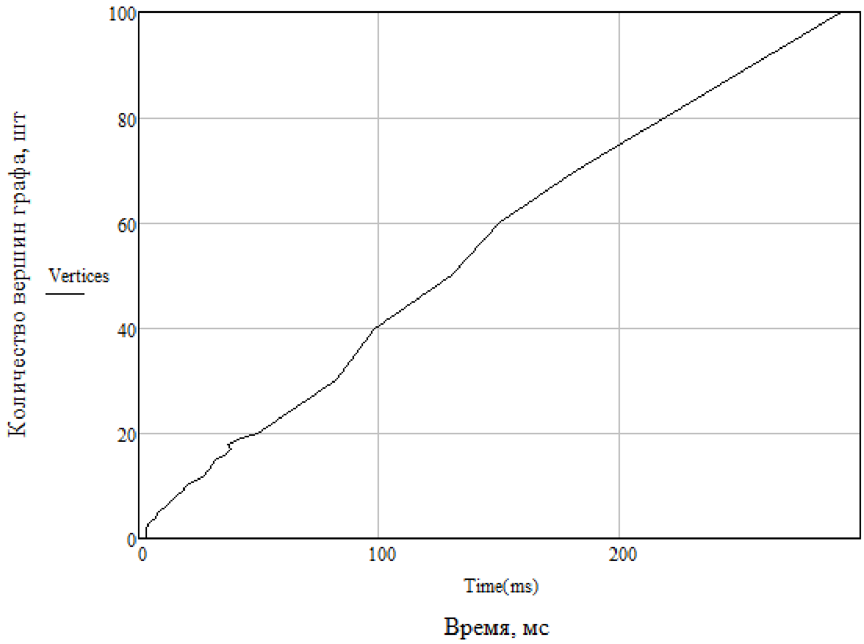

Рисунок 1 – Зависимость времени от количества входных данных на участке от 0 до 100 вершин

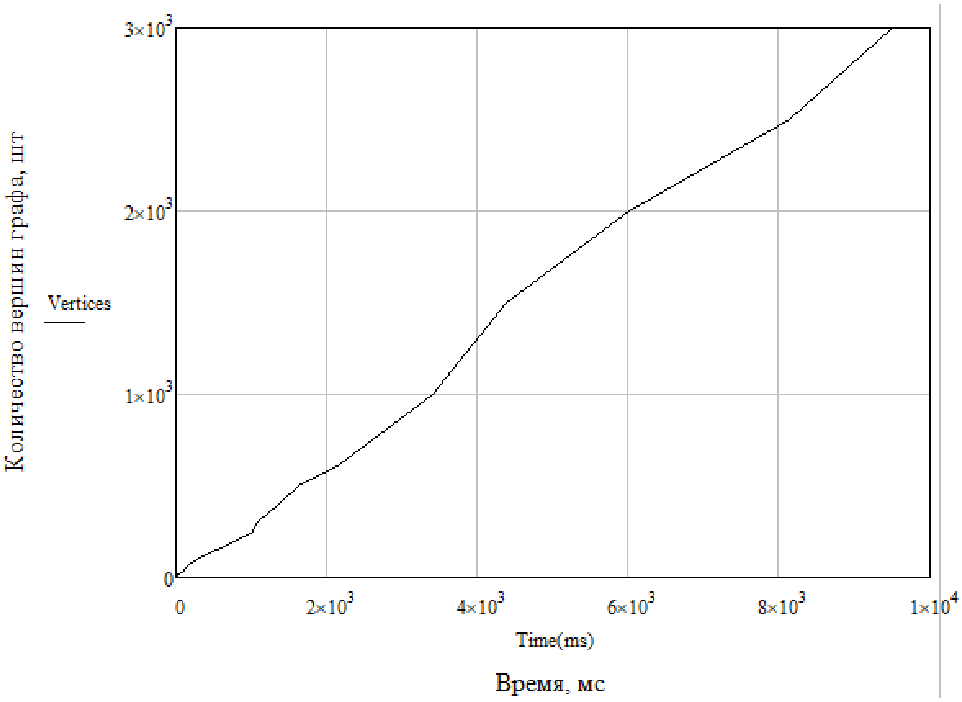

Рисунок 2 – Зависимость времени от количества входных данных

Участок кода для подсчета времени на Рисунке 3.

```
int main() {

    int V = 3000; // Number of vertices in graph
    int E = 6000; // Number of edges in graph
    Graph* graph = createGraph(V, E);
    for (int i = 0; (i+1) <= E; i++) {
        graph->edge[i].src = rand() % V;
        graph->edge[i].end = rand() % V;
        graph->edge[i].weight = rand() % 100;
    }

    auto start = chrono::high_resolution_clock::now();

    KruskalMST(graph);// Function call

    auto end = chrono::high_resolution_clock::now();
    chrono::duration<float> duration = (end - start)*1000; //ms
    cout << "Duration is " << duration.count() << "ms"<< endl;
    return 0;
}
```

Рисунок 3 – Код для измерения времени работы алгоритма

Список литературы

[1] *Кормен, Томас; Чарльз Лейзерсон, Рональд Л. Ривест, Клиффорд Штайн (2009)*. Введение в алгоритмы *(третье изд.). MIT Press. С. 631*

[2] Крускала, JB (1956). «О кратчайшем остовном поддереве графа и задаче коммивояжера». Труды Американского математического общества. 7 (1): 48–50.

[3] Томас Х. Кормен , Чарльз Э. Лейзерсон , Рональд Л. Ривест и Клиффорд Штайн . Введение в алгоритмы, второе издание. MIT Press и McGraw-Hill, 2001. ISBN 0-262-03293-7. Раздел 23.2: Алгоритмы Краскала и Прима, стр. 567–574.

Приложение 1

```cpp
#include <iostream>
#include <cstdlib> // for qsort
#include <chrono>
using namespace std;

// a structure to represent a weighted edge in graph
class Edge {
public:
    int src, end, weight;
};

// a structure to represent a connected, undirected and weighted graph
class Graph {
public:
    // V-> Number of vertices, E-> Number of edges
    int V, E;

    // graph is an array of edges.
    // Since the graph is undirected, the edge
    // from src to end is also edge from end to src.
    // Both are counted as 1 edge here.
    Edge* edge;
};

// Creates a graph with V vertices and E edges
Graph* createGraph(int V, int E) {
    Graph* graph = new Graph;
    graph->V = V;
    graph->E = E;

    graph->edge = new Edge[E];

    return graph;
}

// A structure to represent a subset for Union-find
class subset {
public:
    int parent;
    int rank;
};

// A function to find set of an element i (uses path compression technique)
int find(subset subsets[], int i) {
    // find root and make root as parent of i (path compression)
    if (subsets[i].parent != i) {
        subsets[i].parent = find(subsets, subsets[i].parent);
    }
    return subsets[i].parent;
}

// A function that does Union of two sets of x and y (uses Union by rank)
void Union(subset subsets[], int x, int y) {
    int xroot = find(subsets, x);
    int yroot = find(subsets, y);

    // Attach smaller rank tree under root of high rank tree (Union by Rank)
    if (subsets[xroot].rank < subsets[yroot].rank) {
        subsets[xroot].parent = yroot;
```

```
    } else if (subsets[xroot].rank > subsets[yroot].rank) {
        subsets[yroot].parent = xroot;
    }
    // If ranks are same, then make one as root and increment its rank by one
    else {
        subsets[yroot].parent = xroot;
        subsets[xroot].rank++;
    }
}

// Compare two edges according to their weights.
// Used in qsort() for sorting an array of edges
int myComp(const void* a, const void* b)
{
    Edge* a1 = (Edge*)a;
    Edge* b1 = (Edge*)b;
    return a1->weight > b1->weight;
}

// The main function to construct Minimum Spanning Tree
void KruskalMST(Graph* graph) {
    int V = graph->V;
    Edge result[V]; // This will store the resultant MST
    int e = 0; // An index variable, used for result[]
    int i = 0; // An index variable, used for sorted edges

    // Step 1: Sort all the edges in non-decreasing order of their weight.
    // If we are not allowed to change the given graph,
    // we can create a copy of array of edges
    qsort(graph->edge, graph->E, sizeof(graph->edge[0]),
            myComp);

    // Allocate memory for creating V subsets
    subset* subsets = new subset[(V * sizeof(subset))];

    // Create V subsets with single elements
    for (int v = 0; v < V; ++v) {
        subsets[v].parent = v;
        subsets[v].rank = 0;
    }

    // Number of edges to be taken is equal to V-1
    while (e < V - 1 && i < graph->E) {
        // Step 2: Pick the smallest edge. And increment the index for next iteration
        Edge next_edge = graph->edge[i++];

        int x = find(subsets, next_edge.src);
        int y = find(subsets, next_edge.end);

        // If including this edge does't cause cycle,
        // include it in result and increment the index
        // of result for next edge
        if (x != y) {
            result[e++] = next_edge;
            Union(subsets, x, y);
        }
        // Else discard the next_edge
    }

    // print the contents of result[] to display the
    // built MST
    cout << "Following are the edges in the constructed "
            "MST\n";
    int minimumCost = 0;
```

```
    for (i = 0; i < e; ++i) {
        cout << result[i].src << " -- " << result[i].end << " == " << result[i].weight << endl;
        minimumCost = minimumCost + result[i].weight;
    }
    // return;
    cout << "Minimum Cost Spanning Tree: " << minimumCost << endl;
}


int main() {
    /* Let us create following weighted graph
            10
        0--------1
        |  \     |
       6|   5\   |15
        |     \  |
        2--------3
            4 */
    int V = 4; // Number of vertices in graph
    int E = 5; // Number of edges in graph
    Graph* graph = createGraph(V, E);

    // add edge 0-1
    graph->edge[0].src = 0;
    graph->edge[0].end = 1;
    graph->edge[0].weight = 10;

    // add edge 0-2
    graph->edge[1].src = 0;
    graph->edge[1].end = 2;
    graph->edge[1].weight = 6;

    // add edge 0-3
    graph->edge[2].src = 0;
    graph->edge[2].end = 3;
    graph->edge[2].weight = 5;

    // add edge 1-3
    graph->edge[3].src = 1;
    graph->edge[3].end = 3;
    graph->edge[3].weight = 15;

    // add edge 2-3
    graph->edge[4].src = 2;
    graph->edge[4].end = 3;
    graph->edge[4].weight = 4;

    KruskalMST(graph);// Function call

    return 0;
}
```