Поиск минимального остовного дерева

Сметанина Екатерина

15 мая 2024 г.

Содержание

1 Введение

Алгоритмы поиска минимального остовного дерева являются частью невероятно применимых и распространенных алгоритмов на графах. Самые известные из них алгоритм Прима и алгоритм Краскала.

Fun-fact: Что характерно эти алгоритмы встречалиь при прохождении курса Дискретной математики, поэтому этот проект может оказать значительную помощь при подготовке ко 2ой контрольной(ее переписыванию).

2 Работа над проектом

2.1 Список задач по ТЗ

1. Создание структуры для хранения дерева и функций для смены формата хранения.
2. Реализация алгоритма Прима
3. Реализация алгоритма Краскала
4. Консольный интерфейс
5. Тестирование предыдущих частей
6. Документаци по предыдущим частям
7. *Доп алгоритмы
8. *Тестирование доп алгоритмов
9. *Документация по доп алгоритмам
10. *Графический интерфейс
11. *Документация по граф интерфейсу
12. Соединение всего в единый отчет

2.2 Выбор инструментов

Общее описание

1.Структура.
Ребро хранится ввиде начала, конца и веса.
Граф хранится ввиде матрицы смежности.

2.Сортировка пузырьком.
Работает за N^2

3.Алгоритм Прима.
Строит минимальное остовное дерево (MST). От рандомной вершины строим поддерево, поддерживая приоритетную очередь из вершин еще не включенных в поддерево, в которой ключом для вершины является вес минимального ребра из вершин поддерева в вершину.

4.Алгоритм Краскала.
В граф без ребер добавляем ребра из заранее отсортированного по весу списка. Если наличие ребра уменьшает кол-во компонент связности - добавляем, если не уменьшает - нет.

2.3 Функционал проекта

1.Вводите матрицу смежности/список ребер/список смежности
Создается обьект класса CGraph.
2.Выбираете какаим алгоритмом находить MST
Применяется функция с соответствующм алгоритмом для графа.
3.Любуетесь на матрицу смежности MST

3 Документация по коду

3.1 SEdge

```
//library for I/O from the console
#include<iostream>
//edge
struct SEdge {
        //start
        int a;
        //end
        int b;
        // weight
        int w;
        //constructor
        SEdge(int a = 0, int b = 0, int w = 1) : a(a), b(b), w(w) {}
        //copy constructor
        SEdge(const SEdge& src) : a(src.a), b(src.b), w(src.w) {}
        //deconstructor
        ~SEdge() {}
        //the output operator
        friend std::ostream& operator<<(std::ostream& stream, const SEdge& edge);
};
```

Fun-fact: Так как мы рассматриваем неориентированый граф. Начало и конец ребра ничем не отличаются по смыслу.

```
//edge output operator
std::ostream &operator<<(std::ostream &stream, const SEdge &edge)
{
    std::cout << edge.a << "␣" << edge.b << "␣" << edge.w << std::endl;
}
```

3.2 CGraph общее

```
//graph
class CGraph {
public:
        //constructor
        CGraph(int n = 1) : _n(n), _matr(0) { initMatrix(); };
        //copy constructor
        CGraph(const CGraph& src) : _n(src._n) {
                initMatrix();
                for (int i = 0; i < _n; ++i)
                {
                        for (int j = 0; j < _n; ++j)
                        {
                                _matr[i][j] = src._matr[i][j];
                        }
                }
        }
        //deconstructor
        ~CGraph() { disposeMatrix(); }
        //reading a graph from an adjacency matrix
        void readMatrix();
        //reading a graph from the adjacency list
        void readAdj();
        //reading a graph from a list of edges
        void readEdges();
        //graph output to the adjacency matrix
        void printMatrix();
        //output graph to the adjacency list
        void printAdj();
        //outputting a graph to a list of edges
        void printEdges();
        //output of vertex degrees
        void printPowers();
        //is the graph complete?
        bool isFull();
        //is the graph oriented?
        bool isOriented();
        //is the graph regular?
        bool isRegular();
        //output MST using the algorithm of the Prime
        void printMSTPrima();
        //output MST using the Paint algorithm
        void printMSTKruskal();
private:
        //number of edges
        int edgeNumber();
        //an ascending list of edges
        SEdge* sortedges();
        //allocating memory for the graph adjacency matrix and filling it with 0
        void initMatrix();
        //clearing the memory allocated for the adjacency matrix of the graph
        void disposeMatrix();
        //vertex degree
        int power(int vertex);
        //number of vertices
        int _n;
```

```
        //adjacency matrix
        int** _matr;
};
```

3.3 CGraph read/print

```
void CGraph::readMatrix()
{
        disposeMatrix();
        std::cin >> _n;
        initMatrix();
        for (int i = 0; i < _n; ++i)
        {
                for (int j = 0; j < _n; ++j)
                {
                        std::cin >> _matr[i][j];
                }
        }
}
```

Ожидаемый формат матрицы для считывания:
(число вершин)
(матрица из чисел разделение столбцов пробелами, строк "/n")

```
void CGraph::readAdj()
{
        disposeMatrix();
        std::cin >> _n;
        initMatrix();
        for (int i = 0; i < _n; ++i)
        {
                int m = 0;
                std::cin >> m;
                for (int j = 0; j < m; ++j)
                {
                        int k = 0;
                        std::cin >> k;
                        std::cin >> _matr[i][k - 1];
                }
        }
}
```

Ожидаемый формат списка смежности для считывания:
(число вершин)
..
в iой строке: (число связанных вершин с вершиной i) (j1) (вес ребра i-j1) (j2) (вес ребра i-j2)...
все вершины j связанны с i
..

```
void CGraph::readEdges()
{
        disposeMatrix();
        std::cin >> _n;
        initMatrix();
        int m = 0;
        std::cin >> m;
        for (int k = 0; k < m; ++k)
        {
                int i = 0;
                int j = 0;
```

```
                std::cin >> i >> j;
                std::cin >> _matr[i - 1][j - 1];
        }
```

Ожидаемый формат списка ребер для считывания:
(число вершин)
(число ребер)
..
в іой строке: (начало ребра) (конец ребра) (вес ребра)
..

```
void CGraph::printMatrix()
{
        std::cout << _n << std::endl;
        for (int i = 0; i < _n; ++i)
        {
                for (int j = 0; j < _n; ++j)
                {
                        std::cout << _matr[i][j] << " ";
                }
                std::cout << std::endl;
        }
}

void CGraph::printAdj()
{
        std::cout << _n << std::endl;
        for (int i = 0; i < _n; ++i)
        {
                std::cout << power(i) << " ";
                for (int j = 0; j < _n; ++j)
                {
                        if (_matr[i][j] != 0)
                        {
                                std::cout << j + 1 << " " << _matr[i][j];
                        }
                }
                std::cout << std::endl;
        }
}

void CGraph::printEdges()
{
        std::cout << _n << " ";
        int m = 0;
        for (int i = 0; i < _n; m += power(i), ++i);
        std::cout << m << std::endl;
        for (int i = 0; i < _n; ++i)
        {
                for (int j = 0; j < _n; ++j)
                {
                        if (_matr[i][j] != 0)
                        {
                                std::cout << i + 1 << " " << j + 1 << " " << _matr[i][j]
                        }
                }
                std::cout << std::endl;
```

```
        }
}
```

Вывод производится в тот же формат, что ожидается от ввода.

```
void CGraph::printPowers()
{
        for (int i = 0; i < _n; ++i)
        {
                std::cout << power(i) << " ";
        }
}

bool CGraph::isFull()
{
        if (!isOriented())
        {
                return false;
        }
        int m = 0;
        for (int i = 0; i < _n; m += power(i), ++i);
        return (m * 2 == _n * (_n - 1));
}
```

3.4 CGraph MST

```
void CGraph::printMSTPrima()
{
//checking the orientation of the graph
if (isOriented())
{
std::cout << "Graph should not be oriented for Prima algorithm";
}
        int**ans = new int*[_n];
        bool* used = new bool[_n];
        int* priority = new int[_n];
        int* prev = new int[_n];
        //filling with initial values
        for (int i = 0; i < _n; ++i)
        {
                used[i] = 0;
                priority[i] = 100000;
                prev[i] = -1;
                ans[i] = new int[_n];
                for (int j = 0; j < _n; ++j)
                {
                        ans[i][j] = 0;
                }
        }
        priority[0] = 0;
        for (int i = 0; i < _n; ++i)
        {
                int v = -1;
                //looking for a priority vertex
                for (int j = 0; j < _n; ++j)
                {
                        if (!used[j] && (v == -1 || priority[j] < priority[v]))
                                v = j;
                }
                if (priority[v] == 100000)
                {
                        std::cout << "No MST!";
                        return;
                }
                //adding it to the subtree
                used[v] = true;
                if (prev[v] != -1)
                {
                        ans[v][prev[v]] = _matr[v][prev[v]];
                        ans[prev[v]][v] = _matr[v][prev[v]];
                }
                //changing priorities based on the new subtree
                for (int to = 0; to < _n; ++to)
                {
                        if (_matr[v][to] < priority[to])
                        {
                                priority[to] = _matr[v][to];
                                prev[to] = v;
                        }
                }
```

```
        }
        //output of the MST proximity matrix
        for (int i = 0; i < _n; ++i)
        {
                std::cout << "\n";
                for (int j = 0; j < _n; ++j)
                {
                        std::cout << ans[i][j] << " ";
                }
        }
        for (int i = 0; i < _n; ++i)
        {
                delete[] ans[i];
        }
        delete[] ans;
        delete[] used;
        delete[] prev;
        delete[] priority;
}
```

```
void CGraph::printMSTKruskal()
{
	if (isOriented())
	{
		std::cout << "Graph should not be oriented for Kruskal alghoritm";
	}
	int m = edgeNumber();
	SEdge* edges = sortedEdges();
	SEdge* ans = new SEdge[_n - 1];
	int ansTmp = 0;
	int* treeColor = new int[_n];
	for (int i = 0; i < _n; ++i)
	{
		treeColor[i] = i;
	}
	for (int i = 0; i < m; ++i)
	{
		int a = edges[i].a;
		int b = edges[i].b;
		int w = edges[i].w;
		if (treeColor[a] != treeColor[b])
		{
			ans[ansTmp] = SEdge(a, b, w);
			ansTmp++;
			int oldColor = treeColor[b];
			int newColor = treeColor[a];
		//joining trees
			for (int j = 0; j < _n; ++j)
			{
				if (treeColor[j] == oldColor)
				{
					treeColor[j] = newColor;
				}
			}
		}
	}
	//converting and outputting MST
	int** ansMatr = new int*[_n];
	for (int i = 0 ; i < _n; ++i)
	{
		ansMatr[i] = new int[_n];
		for (int j = 0; j < _n; ++j)
		{
			ansMatr[i][j] = 0;
		}
	}

	for (int i = 0; i < _n - 1; ++i)
	{
		ansMatr[ans[i].a][ans[i].b] = ans[i].w;
		ansMatr[ans[i].b][ans[i].a] = ans[i].w;
	}

	for (int i = 0 ; i < _n; ++i)
	{
		std::cout << "\n";
```

```
                for (int j = 0; j < _n; ++j)
                {
                        std::cout << ansMatr[i][j] << " ";
                }
        }
        for (int i = 0; i < _n; ++i)
        {
                delete[] ansMatr[i];
        }
        delete[] ansMatr;
}
```

3.5 Other

```
int CGraph::edgeNumber()
{
        int m = 0;
        for (int i = 0; i <_n; ++i)
        {
                for (int j = i + 1; j < _n; ++j)
                {
                        if (_matr[i][j] != 0)
                        {
                                m++;
                        }
                }
        }
        return m;
}
```

Кол-во ненулевых ребер выше главной диагонали

```
SEdge* CGraph::sortedEdges()
{
        int m = edgeNumber();
        SEdge* ans = new SEdge[m];
        int tmp = 0;
        for (int i = 0; i <_n; ++i)
        {
                for (int j = i + 1; j < _n; ++j)
                {
                        if (_matr[i][j] != 0)
                        {
                                ans[tmp] = SEdge(i, j, _matr[i][j]);
                                tmp++;
                        }
                }
        }
        SEdge tmpEdge = SEdge();
        for (int i = 0; i < m; ++i)
        {
                for (int j = 0; j < m - 1; ++j)
                {
                        if (ans[j].w > ans[j + 1].w)
                        {
                                tmpEdge = ans[j];
                                ans[j] = ans[j + 1];
                                ans[j + 1] = tmpEdge;
                        }
                }
        }
        return ans;
}
```

Извлечение ребер и сортировка пузырьком.

```
void CGraph::initMatrix()
{
        _matr = new int* [_n];
        for (int i = 0; i < _n; ++i)
        {
```

```
                _matr[i] = new int[_n];
                for (int j = 0; j < _n; ++j)
                {
                        _matr[i][j] = 0;
                }
        }
}

void CGraph::disposeMatrix()
{
        for (int i = 0; i < _n; ++i)
        {
                delete[] _matr[i];
        }
        delete[] _matr;
}

int CGraph::power(int vertex)
{
        int pw = 0;
        for (int j = 0; j < _n; ++j)
        {
                pw += (_matr[vertex][j] > 0);
        }
        return pw;
}


bool CGraph::isOriented()
{
        for (int i = 0; i < _n; ++i)
        {
                if (_matr[i][i] == 1)
                {
                        return false;
                }
        }
        for (int i = 0; i < _n; ++i)
        {
                for (int j = i + 1; j < _n; ++j)
                {
                        if (_matr[i][j] != _matr[j][i])
                        {
                                return true;
                        }
                }
        }
        return false;
}
```

4 Выводы

Несмотря на кажущуюся уневерсальность класса CGraph он требует больших изменений и крайней степени внимательности при адаптации его к новым задачам, особенно при переходах взвешанны/невзвешанный и ориентированный/неориентированный. Весьма полезным может быть написание общего класса и создание у него наследников для всех возможных комбинаций этих свойств.