Министерство образования и науки Российской Федерации
Федеральное государственное автономное образовательное учреждение
высшего образования «Национальный исследовательский Нижегородский
государственный университет им. Н.И. Лобачевского»

Институт Информационных технологий, математики и механики

Отчёт по учебной практике

Алгоритм Дейкстры на D-куче

Выполнил:
студент гр. 381606-3

Каганов Д.А.

Проверил:
к.т.н., доцент МОСТ ИИТММ

Кустикова В.Д.

Нижний Новгород
2018 г.

Содержание

Введение

Приоритетная очередь — это абстрактный тип данных, значениями которого являются взвешенные множества. Множество называется взвешенным, если каждому его элементу однозначно соответствует вещественное число, называемое ключом или весом. Приоритетная очередь естественным образом применяется в таких задачах, как сортировка элементов массива во внутренней памяти (пирамидальная сортировка), отыскание в графе минимального связывающего дерева (алгоритм Крускала), отыскание кратчайших путей от заданной вершины графа до его остальных вершин (алгоритм Дейкстры) и при алгоритмизации многих других задач. Вид дерева и способ его представления в памяти компьютера подбирается в зависимости от тех операций, которые предполагается выполнять над множеством и от того, насколько эти операции сказываются на суммарной трудоемкости алгоритма. Представления приоритетной очереди с помощью – d-кучи ($d \geq 2$), основано на использовании так называемых завершенных d-арных деревьев.

Постановка задачи

1) Разработать статические библиотеки, реализующие следующие структуры данных:
    a) d-кучу;
    b) алгоритм Дейкстры;
    c) разделенные множества;
    d) приоритетную очередь, основанную на d-куче;
2) Написать приложение для демонстрации работы приоритетной очереди, основанной на d-куче (алгоритм Дейкстры):
    a) входные данные - связный неориентированный взвешенный граф со стартовой вершиной;
    b) выходные данные:
        i) список вершин;
        ii) список кратчайших путей до каждой вершины графа;
        iii) список предшествующий вершин.

Руководство пользователя

1) Запустите программу.
2) В появившемся окне введите кол-во вершин и ребер.
3) Введите в консоль стартовую вершину для алгоритма Дейкстры
4) Программа выведет кол-во вершин и стартовую вершину
5) Программа выведет матрицу взвешенных ребер и расстояний от заданной точки
6) Программа выведет пути до всех вершин (вершина, расстояние от заданной, предшествующая вершина)

Руководство программиста

Описание структуры программы

1) ../include
    a) Dejkstra.h
       Содержит описание класса Dejkstra
    b) DHeap.h
       Содержит описание класса DHeap
    c) Graph.h
       Содержит описание класса Graph
    d) PriorityQueue.h
       Содержит описание класса PriorityQueue
2) ../sample
    a) GenerateGraph.cpp
       Содержит основной код программы
3) ../src
    a) Dejkstra.h
       Содержит реализацию класса Dejkstra
    b) DHeap.h
       Содержит реализацию класса DHeap
    c) Graph.h
       Содержит реализацию класса Graph
    d) PriorityQueue.h
       Содержит реализацию класса PriorityQueue и DHeapPriorityQueue
4) ../build
   Содержит директорию с решением и проектом для MS Visual Studio

Описание структур данных

Шаблонный класс Dejkstra

```
class DataFloat : public Data {
public:
	DataFloat(int s, float dist);
	int s;
};
```

```
class Dejkstra {
public:
	static void dejkstra(Graph *&graph, int s, float *&distance, int *&up);
};
```

Описание методов:

1. dejkstra - алгоритм Дэйкстры

Шаблонный класс DHeap

```
typedef int dataType;

class Data {
public:
	float priorities;
};

class DHeap {
protected:
	Data **keys;
	int d;
	int idx;
public:
	DHeap(int d);
	DHeap(const DHeap &heap);
	~DHeap();

	void Add(Data *&key);
	void AddSet(Data **key, int num);
	Data* Delete();
	Data* Remove(int i);

	void Transpose(int i, int j);
	void Surfacing(int i);
	void Sinking(int i);
	void Hilling();
	int IsFull();
	int IsEmpty();

private:
	int MinChild(int i);
};
```

Описание методов:

1. Transpose - транспонирование
2. Surfacing - всплытие узла
3. Sinking - погружение
4. Hilling - окучивание
5. MinChild - погружение узла i

Шаблонный класс Graph

```
class Edge {
public:
	int Ne;
	int Ke;
	float W;

	Edge(int Ne, int Ke, float W);
};

class Graph
{
private:
	int n;
	int m;
	int m_cur;
	Edge** edges;
	int* vertices;
public:
	Graph(int n);
	Graph(int n, int m);
	~Graph();

	void Generate(float minRange, float maxRange);
	void AddEdge(int Ne, int Ke, float weight);
	void DelEdge(int Ne, int Ke);
	int GetVerticesNum();
	int GetEdgeSize();
	int GetRealSize();
	Edge** GetEdgeSet();
	Edge*  GetEdge(int i);
	float GetWeight(int Ne, int Ke);
	void PrintList();
private:
	void GenerateVertices(int &Ne, int &Ke);
	float GenerateWeight(float minRange, float maxRange);
	void Clean();
	int FindEdge(int Ne, int Ke);
};
```

Описание методов:

1. Generate - генерация графа в рандомном режиме
2. AddEdge - добавление ребра
3. DelEdge - удаление ребра
4. GetVerticesNum - число вершин
5. GetEdgeSize - длина ребер
6. GetRealSize - реальное число
7. GetEdgeSet - массив взвешенных ребер
8. GetEdge - возвращает заданное ребро
9. GetWeight - вес ребра с заданными вершинами
10. PrintList - вывод
11. GenerateVertices - генерация вершин
12. GenerateWeight - генерация веса вершин
13. Clean - удаление всех взвешенных ребер
14. FindEdge - поиск ребра по заданным вершинам

Описание алгоритмов

Дан взвешенный ориентированный граф G(V,E) без дуг отрицательного веса. Найти кратчайшие пути от некоторой вершины a графа G до всех остальных вершин этого графа.

- n — множество вершин графа
- m — множество рёбер графа
- minRange — минимальный вес ребра
- maxRange — максимальный вес ребра
- s — вершина, расстояния от которой ищутся

В простейшей реализации для хранения чисел $d[i]$ можно использовать массив чисел, а для хранения принадлежности элемента множеству U — массив булевых переменных.

В начале алгоритма расстояние для начальной вершины полагается равным нулю, а все остальные расстояния заполняются большим положительным числом (бóльшим максимального возможного пути в графе). Массив флагов заполняется нулями. Затем запускается основной цикл.

На каждом шаге цикла мы ищем вершину v с минимальным расстоянием и флагом равным нулю. Затем мы устанавливаем в ней флаг в 1 и проверяем все соседние с ней вершины u. Если в них (в u) расстояние больше, чем сумма расстояния до текущей вершины и длины ребра, то уменьшаем его. Цикл завершается, когда флаги всех вершин становятся равны 1, либо когда у всех вершин с флагом 0 d[i]=∞. Последний случай возможен тогда и только тогда, когда граф G несвязный.

```
DataFloat::DataFloat(int s, float dist)
{
	this->s = s;
	priorities = dist;
}

void Dejkstra::dejkstra(Graph *&graph, int s, float *&distance, int *&up)
{
	int n = graph->GetVerticesNum();
	int m = graph->GetRealSize();
	if ((s < 0) || (s >= n))
		throw "Dejkstra: Invalid start vertex";

	Data** dist = new Data*[n];
	up = new int[n];

	PriorityQueue *queue = QueueFactory::createQueue(static_cast<QueueID>(0));

	for (int i = 0; i < n; i++) {
		up[i] = i;
		dist[i] = new DataFloat(i, FLT_MAX);
		if (i == s)
			dist[s]->priorities = 0;
		queue->Push(dist[i]);
	}

	Edge** edges = graph->GetEdgeSet();
	while (!queue->IsEmpty())
	{
		int vConsidered = ((DataFloat*)queue->Pop())->s;
		float delta;

		for (int i = 0; i < m; i++)
		{
			int vIncident = -1;
			if (edges[i]->Ke == vConsidered)
				vIncident = edges[i]->Ne;
			if (edges[i]->Ne == vConsidered)
				vIncident = edges[i]->Ke;
			if (vIncident == -1) continue;
```

```
                    float way = dist[vConsidered]->priorities +
graph->GetWeight(vConsidered, vIncident);
                    delta = dist[vIncident]->priorities - way;
                    if (delta > 0)
                    {
                           dist[vIncident]->priorities = way;
                           up[vIncident] = vConsidered;
                           queue->Refresh();
                    }
             }
      }

      distance = new float[n];
      for (int i = 0; i < n; i++)
            distance[i] = dist[i]->priorities;

      for (int i = 0; i < n; i++)
            delete dist[i];
      delete[]dist;
      delete queue;
}
```

Заключение

В рамках лабораторной работы мы разработали статические библиотеки, реализующие следующие структуры данных: d-куча, алгоритм Дейкстры, приоритетная очередь, основанную на d-куче. Написали приложение для демонстрации работы приоритетной очереди, основанной на d-куче (алгоритм Дейкстры).