МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ 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 лабораторной работе

«Алгоритм Дейкстры поиска кратчайших путей из одной вершины»

Выполнил:

Студент группы 381706-2

Исаев И.А.

Проверил:

Доцент кафедры МОСТ, кандидат технических наук, Сысоев А.В.

Нижний Новгород

2019

Содержание

Введение

Поиск кратчайших путей – одна из важнейших задач в теории графов. Она является достаточно значимой и применяется в различных областях: картографические сервисы, протоколы маршрутизации, сети дорог.

Сегодня известно множество алгоритмов решения данной задачи. Кратчайший рассматривается с помощью математической модели, называемой графом.

Граф – это абстрактный математический объект, представляющий собой множество вершин графа и набор рёбер, которые соединяют пары вершин между собой.

Граф называется **ориентированным**, если множество рёбер E – упорядоченное. В случае **неориентированного** оно неупорядоченное.

Граф называется **взвешенным**, если каждому ребру из множества рёбер E поставлено в соответствие некоторое значение, называемое весом графа.

Одним из алгоритмов решения задачи поиска кратчайших путей является алгоритм Дейкстры.

Алгоритм Дейкстры – это алгоритм на графах, предложенный нидерландским учёным Эдсгером Дейкстрой в 1959 году. Этот алгоритм позволяет найти пути из одной из вершин взвешенного графа (ориентированного и неориентированного), в котором нет отрицательных весов.

Постановка задачи

В рамках данной лабораторной работы ставится задача программной реализации библиотеки, которая включает в себя:

1. Генерация случайного взвешенного графа и, следовательно, способ его хранения.
2. Последовательный алгоритм Дейкстры.
3. Параллельный алгоритм Дейкстры.

Данная библиотека должна быть представлена отдельным модулем и иметь набор тестов, которые подтверждают корректность данной реализации.

Метод решения

Пусть дан взвешенный граф с n вершинами и m рёбрами. Веса всех рёбер неотрицательны. Указана некоторая стартовая вершина s. Требуется найти длины кратчайших путей из s во все остальные вершины.

Заведём массив d, в котором для каждой вершины u будем хранить текущую длину $d[u]$ кратчайшего пути из s в u. Изначально $d[s] = 0$, а для всех остальных вершин эта величина равна бесконечности.

Также заведём массив меток v, в котором для каждой вершины u будем хранить, помечена она или нет. Изначально $v[u] = false$ для всех вершин, то есть все они не помечены.

Сам алгоритм Дейкстры состоит из n итераций, на очередной итерации выбирается вершина u с наименьшей величиной $d[u]$ из непомеченных.

$$d[u] = \min_{p:v[p]=false} d[p]$$

Выбранная таким образом вершина будет считаться помеченной. Далее, на текущей итерации, просматриваются все рёбра, исходящие из вершины u, и для каждой вершины t вычисляется $d[t]$:

$$d[t] = \min(d[t], d[u] + len)\ , len - \text{расстояние от } u \text{ до } t$$

На этом итерация завершается и переходит к следующей. Итогом будет массив d, в котором хранятся все кратчайшие расстояния из вершины s

Такую реализацию алгоритма Дейкстры называют «алгоритмом Дейкстры с метками».

Схема распараллеливания

Граф в данной реализации представлен в виде матрицы смежности, где на пересечении j-го столбца и i-ой строки расположен вес ребра пары вершин (i, j).

Вследствие того, что граф может быть неориентированным, матрица смежности может быть несимметричной, поэтому необходимо разбивать матрицу на блоки из столбцов, которые будет получать, каждый процесс. Это повышает трудоёмкость задачи, так как матрицу сперва нужно транспонировать.

Каждый процесс получает блок из N/P строк транспонированной матрицы, где N – число вершин графа, P – число запущенных процессов. Первый же процесс, в свою очередь, получает первые $N/P + N\%P$ строки транспонированной матрицы.

Дальнейшие вычисления на каждом процессе схожи с тем, что предлагает последовательный алгоритм, а именно поиск минимального значения, проход по рёбрам, только каждый процесс выполняет это своим блоком данных, при этом сравнивая своё локальное полученное значение с другими процессами с помощью операции MPI_Allreduce.

Описание программной реализации

В программе реализован класс хранения графа. Как уже было упомянуто, граф представляет собой матрицу смежности. В классе Graph определенны операции:

- getAdjacency – получить матрицу смежности графа
- setAdjacency (const std::vector<int> _adj) – присвоить матрицу смежности графу

Рассмотрим функции, реализованные в модуле:

- createRandomGraph(const int& n, GraphType type = UNDIRECTED) – генерация случайного графа, где первый параметр – число вершин графа, второй – тип графа (ориентированный или неориентированный)
- minDistanceVertex(const std::vector<int>& dist, const std::vector<int>& marker) – поиск вершины u с минимальным значение dist[u], где marked[u] = false
- getTransposeAdjacency(const std::vector<int>& mat) – функция, которая транспонирует матрицу смежности.
- dijkstraSequential(const Graph& graph, const int& src) – последовательный алгоритм, принимает в качестве параметров граф и вершину-источник.
- dijkstraParallel(const Graph& graph, const int& src) – параллельный алгоритм, принимает в качестве параметров граф и вершину-источник.

А теперь рассмотрим последнюю функцию подробнее:

1. Сначала происходит вызов MPI_Comm_rank и MPI_Comm_size, чтобы каждый процесс узнал свой ранг и число всех процессов соответственно.
2. Далее идёт транспонирование матрицы и распределение уже её строк между процессорами посредством операции MPI_Scatter, при этом первый процесс получает ещё остаток.
3. Векторы расстояний и меток заранее инициализируются к первой итерации алгоритма.
4. Запускается основной цикл алгоритма. С помощью функции minDistanceVertex каждый процесс находит локальную вершину с минимальным значением вектора расстояний.
5. С помощью операцию MPI_Allreduce собираются все полученные локальные вершины и выбирается с самым минимальным значением.

6. Эта вершина становится помеченной, т.е. в векторе locmarked этой вершине присваивается значение true.
7. Производится обход по рёбрам в каждом процессе с попыткой улучшить значение locdist.
8. После завершения основного цикла с помощью MPI_Send и MPI_Recv данные с каждого процесса собираются первым процессом, и функция завершает работу.

Подтверждение корректности

Для подтверждения корректности в программе представлено 5 тестов, разработанных с помощью Google C++ Testing Framework.

Эти тесты представляют из себя проверку на невозможность работы функций с неверными данными, проверку последовательного и параллельного алгоритмов на идентичность результата, а также проверка работы алгоритма по заранее известному набору данных.

Успешное прохождение всех тестов подтверждает корректность данной программной реализации.

Результаты экспериментов

Эксперимент для оценки эффективности параллельного алгоритма Дейкстры в сравнении с последовательным проводился на компьютере со следующей конфигурацией:

- Процессор: Intel Core i5 3210M 3.1GHz (2 ядра, 4 потока)
- Оперативная память: 6Gb DDR3
- Операционная система: Manjaro Linux 18.1.5

Число вершин графа – 9000.

Результаты приведены в Таблице 1.

Количество процессов	Время работы последовательного алгоритма, сек	Время работы параллельного алгоритма, сек	Ускорение
1	1.63623	2.23967	0.7305
2	1.6226	1.56289	1.0382
3	1.64408	1.43924	1.1423
4	1.65034	1.34533	1.2267
5	1.65527	1.54455	1.0716

Таблица 1. Результаты вычислительного эксперимента.

Как можно заметить, ускорение относительно последовательного алгоритма с увеличением числа процессов невелико, так как трудоёмкость алгоритма выше из-за транспонирования матрицы смежности. Также можно заметить, что на 4 процессах ускорение выше, чем на 2, хотя число физических ядер всего 2. Это обусловлено тем, что в системе на самом деле 4 логических ядра, и алгоритм Дейкстры состоит из повторяющихся операций, которые в инструкциях вычислительного конвейера кэшируются.

При числе процессов больше 4, ускорение будет снижаться ввиду недостатка вычислительных ресурсов, так как некоторые процессы будут простаивать.

Заключение

Таким образом, в результате лабораторной работы была разработана программа, реализующая последовательный и параллельный алгоритмы Дейкстры. Задача была достигнута, и из проведённого эксперимента можно сделать вывод, что параллельный алгоритм работает действительно быстрее, чем последовательный, следовательно, он является более предпочтительным.

Литература

1. Алгоритм Дейкстры. URL: en.wikipedia.org/wiki/Dijkstra%27s_algorithm
2. Parallel all-pairs shortest path algorithm. URL: en.wikipedia.org/wiki/Parallel_all-pairs_shortest_path_algorithm
3. Нахождение кратчайших путей из заданной вершины. URL: e-maxx.ru/algo/dijkstra

Приложение

dijkstra_alg.h

```
// Copyright 2019 Isaev Ilya
#ifndef MODULES_TASK_3_ISAEV_DIJKSTRA_ALG_DIJKSTRA_ALG_H_
#define MODULES_TASK_3_ISAEV_DIJKSTRA_ALG_DIJKSTRA_ALG_H_
#include <vector>

const int infinity = 999999999;
enum GraphType{UNDIRECTED, DIRECTED};

class Graph {
 private:
   std::vector<int> adjacency;
   GraphType type;
 public:
   explicit Graph(const int& vsize = 1, GraphType _type = UNDIRECTED);
   std::vector<int> getAdjacency() const {return adjacency;}
   GraphType getType() const {return type;}
   void setAdjacency(const std::vector<int> _adj);
};

std::vector<int> createRandomGraph(const int& n, GraphType type = UNDIRECTED);
int minDistanceVertex(const std::vector<int>& dist, const std::vector<int>& marker);
std::vector<int> dijkstraSequential(const Graph& graph, const int& src);
std::vector<int> dijkstraParallel(const Graph& graph, const int& src);
std::vector<int> getTransposeAdjacency(const std::vector<int>& mat);

#endif  // MODULES_TASK_3_ISAEV_DIJKSTRA_ALG_DIJKSTRA_ALG_H_
```

dijkstra_alg.cpp

```
// Copyright 2019 Isaev Ilya

#include "../../../modules/task_3/isaev_dijkstra_alg/dijkstra_alg.h"

#include <mpi.h>

#include <algorithm>

#include <vector>

#include <ctime>

#include <random>

#include <iostream>
```

```
Graph::Graph(const int& vsize, GraphType _type): type(_type) {
    if (vsize <= 0)
        throw -1;
    adjacency.resize(vsize*vsize);
}


void Graph::setAdjacency(const std::vector<int> _adj) {
    if (_adj.size() != adjacency.size())
        throw -1;
    type = UNDIRECTED;
    int n = sqrt(static_cast<int>(_adj.size()));
    if (n*n != static_cast<int>(_adj.size()))
        throw -1;
    for (int i = 0; i < n; ++i) {
        for (int j = i; j < n; ++j) {
            if (_adj[j+i*n] != _adj[i+j*n]) {
                type = DIRECTED;
                break;
            }
        }
    }
    adjacency = _adj;
}
```

```
std::vector<int> createRandomGraph(const int& n, GraphType type) {
    if (n <= 0)
        throw -1;
    std::default_random_engine random;
    random.seed(static_cast<unsigned int>(time(0)));
    std::vector<int> adjacency = std::vector<int>(n*n);
    for (int i = 0; i < n; ++i) {
        for (int j = (type == 0) ? i : 0; j < n; ++j) {
            if (i == j) {
                adjacency[j+i*n] = 0;
            } else {
                adjacency[j+i*n] = random() % 100;
                if (adjacency[j+i*n] == 0) {
                    adjacency[j+i*n] = infinity;
                }
                if (type == 0) {
                    adjacency[i+j*n] = adjacency[j+i*n];
                }
            }
        }
    }
    return adjacency;
}
```

```
std::vector<int> dijkstraSequential(const Graph& graph, const int& src) {
    int n = sqrt(static_cast<int>(graph.getAdjacency().size()));
    if (src < 0 || src >= n)
        throw -1;
    std::vector<int> adj = graph.getAdjacency();
    std::vector<int> dist(n, infinity);
    std::vector<int> marked(n, 0);
    dist[src] = 0;
    for (int i = 0; i < n-1; ++i) {
        int vertex = minDistanceVertex(dist, marked);
        marked[vertex] = 1;
        for (int j = 0; j < n; ++j) {
            if (!marked[j] && adj[j+vertex*n] != infinity
                && (dist[vertex] + adj[j+vertex*n] < dist[j])) {
                dist[j] = dist[vertex] + adj[j+vertex*n];
            }
        }
    }
    return dist;
}

std::vector<int> dijkstraParallel(const Graph& graph, const int& src) {
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
```

```
int n = sqrt(static_cast<int>(graph.getAdjacency().size()));
if (src < 0 || src >= n)
   throw -1;
int delta = n/size;
int mod = n%size;
int localmod = (rank == 0) ? n%size : 0;
std::vector<int> sendgraph(n*n);
std::vector<int> loc_adjacency(delta*n);

if (rank == 0) {
   if (graph.getType() == 0)
      sendgraph = graph.getAdjacency();
   else
      sendgraph = getTransposeAdjacency(graph.getAdjacency());
}
if (delta != 0)
   MPI_Scatter(&sendgraph[mod*n], delta*n, MPI_INT, &loc_adjacency[0], delta*n, MPI_INT, 0, MPI_COMM_WORLD);
if (rank == 0) {
   loc_adjacency.insert(loc_adjacency.begin(), sendgraph.begin(), sendgraph.begin()+mod*n);
}
std::vector<int> locdist(delta+localmod, infinity);
std::vector<int> locmarked(delta+localmod, 0);

for (int i = 0; i < delta+localmod; ++i) {
```

```
    locdist[i] = loc_adjacency[src + i*n];
    if (locdist[i] == 0)
        locmarked[i] = 1;
}

struct {
    int value;
    int index;
} in, out;

for (int count = 0; count < n-1; ++count) {
    int locindex = minDistanceVertex(locdist, locmarked);
    if (locindex != -1) {
        in.index = mod-localmod + locindex+rank*(delta+localmod);
        in.value = locdist[locindex];
    } else {
        in.index = -1;
        in.value = infinity;
    }
    MPI_Allreduce(&in, &out, 1, MPI_2INT, MPI_MINLOC, MPI_COMM_WORLD);
    if (out.index == -1)
        break;
    if (in.index == out.index && locindex != -1) {
        locmarked[locindex] = 1;
    }
```

```
            for (int i = 0; i < delta + localmod; ++i) {
                if (!locmarked[i] && loc_adjacency[out.index+i*n] != infinity
                && (out.value + loc_adjacency[out.index+i*n] < locdist[i])) {
                    locdist[i] = out.value+loc_adjacency[out.index+i*n];
                }
            }
        }
        std::vector<int> res;
        if (rank == 0) {
            MPI_Status status;
            std::vector<int> recv(delta);
            res.insert(res.begin(), locdist.begin(), locdist.end());
            for (int i = 1; i < size; ++i) {
                MPI_Recv(&recv[0], delta, MPI_INT, i, 2, MPI_COMM_WORLD, &status);
                res.insert(res.end(), recv.begin(), recv.end());
            }
        } else {
            MPI_Send(&locdist[0], delta, MPI_INT, 0, 2, MPI_COMM_WORLD);
        }
        return res;
    }

    int minDistanceVertex(const std::vector<int>& dist, const std::vector<int>& marker) {
        int minvalue = infinity;
        int res = -1;
```

```
    for (int i = 0; i < static_cast<int>(dist.size()); ++i) {
        if (marker[i] == 0 && dist[i] <= minvalue) {
            minvalue = dist[i];
            res = i;
        }
    }
    return res;
}

std::vector<int> getTransposeAdjacency(const std::vector<int>& mat) {
    int n = sqrt(static_cast<int>(mat.size()));
    std::vector<int> tmp(n*n);
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            tmp[i+j*n] = mat[j+i*n];
        }
    }
    return tmp;
}
```

main.cpp

```
// Copyright 2019 Isaev Ilya
#include <gtest-mpi-listener.hpp>
#include <gtest/gtest.h>
#include <vector>
#include <algorithm>
#include "./dijkstra_alg.h"

TEST(Dijkstra_Alg, CAN_CREATE_GRAPH) {
   int rank, size;
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   if (rank == 0) {
      ASSERT_NO_THROW(Graph(5));
   }
}

TEST(Dijkstra_Alg, THROWS_ON_NEGATIVE_V_SIZE) {
   int rank, size;
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   if (rank == 0) {
      ASSERT_ANY_THROW(Graph(-5));
   }
}
```

```
    TEST(Dijkstra_Alg,
SEQUENTIAL_AND_PARALLEL_HAVE_SAME_ANSWER_UNDIRECTED) {
        int rank, size;
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Comm_size(MPI_COMM_WORLD, &size);
        int nv = 500;
        Graph graph(nv);
        std::vector<int> seqres;
        std::vector<int> parres;
        if (rank == 0) {
            graph.setAdjacency(createRandomGraph(nv));
            seqres = dijkstraSequential(graph, 0);
        }
        parres = dijkstraParallel(graph, 0);
        if (rank == 0) {
            ASSERT_EQ(seqres, parres);
        }
    }


    TEST(Dijkstra_Alg,
SEQUENTIAL_AND_PARALLEL_HAVE_SAME_ANSWER_DIRECTED) {
        int rank, size;
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Comm_size(MPI_COMM_WORLD, &size);
        int nv = 500;
```

```
    Graph graph(nv);
    std::vector<int> seqres;
    std::vector<int> parres;
    if (rank == 0) {
        graph.setAdjacency(createRandomGraph(nv, DIRECTED));
        seqres = dijkstraSequential(graph, 0);
    }
    parres = dijkstraParallel(graph, 0);
    if (rank == 0) {
        ASSERT_EQ(seqres, parres);
    }
}

TEST(Dijkstra_Alg, THROWS_ON_WRONG_SOURCE) {
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    if (rank == 0) {
        Graph graph(5);
        ASSERT_ANY_THROW(dijkstraParallel(graph, 7));
    }
}

int main(int argc, char** argv) {
    ::testing::InitGoogleTest(&argc, argv);
```

```
    MPI_Init(&argc, &argv);

    ::testing::AddGlobalTestEnvironment(new GTestMPIListener::MPIEnvironment);
    ::testing::TestEventListeners& listeners =
        ::testing::UnitTest::GetInstance()->listeners();

    listeners.Release(listeners.default_result_printer());
    listeners.Release(listeners.default_xml_generator());

    listeners.Append(new GTestMPIListener::MPIMinimalistPrinter);
    return RUN_ALL_TESTS();
}
```