Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное учреждение высшего образования

«Нижегородский государственный университет им. Н.И. Лобачевского»

**Институт информационных технологий, математики и механики**

**Отчет**

по лабораторной работе №3

**Поиск кратчайших путей из одной вершины (алгоритм Дейкстры)**

**Выполнил:**

студент группы 381608

Мезина М.Д.

Нижний Новгород

2018

# **План работы**

Оглавление

[**План работы** 2](#_Toc533288591)

[**Постановка задачи** 3](#_Toc533288592)

[**Метод решения** 4](#_Toc533288593)

[**Схема распараллеливания** 6](#_Toc533288594)

[**Описание программной реализации** 7](#_Toc533288595)

[**Подтверждение корректности** 9](#_Toc533288596)

[**Результаты экспериментов** 10](#_Toc533288597)

[**Вывод** 12](#_Toc533288598)

# **Постановка задачи**

Дан неориентированный взвешенный граф с *num\_vert* вершинами и *num\_edge* рёбрами. Веса всех рёбер неотрицательны. Указана некоторая стартовая вершина *start*. Требуется найти длины кратчайших путей из вершины *start* во все остальные вершины, а также предоставить способ вывода самих кратчайших путей, используя алгоритм Дейкстры.

# **Метод решения**

Заведём массив *distance[]*, в котором для каждой вершины *vert* будем хранить текущую длину *distance[vert]* кратчайшего пути из *start* в *vert*. Изначально *distance[start]=0*, а для всех остальных вершин эта длина равна бесконечности.

Кроме того, для каждой вершины *vert* будем хранить, помечена она ещё или нет, т.е. заведём булевский массив *was[]*. Изначально все вершины не помечены, т.е.

Сам алгоритм Дейкстры состоит из *num\_vert* итераций. На очередной итерации выбирается вершина *vert* с наименьшей величиной *distance[vert]* среди ещё не помеченных, т.е.:

(Понятно, что на первой итерации выбрана будет стартовая вершина *start*)

Выбранная таким образом вершина *vert* отмечается помеченной. Далее, на текущей итерации, из вершины *vert* производятся **релаксации**: просматриваются все рёбра *(vert, to)*, исходящие из вершины *vert*, и для каждой такой вершины *to* алгоритм пытается улучшить значение *distance[to]*. Пусть длина текущего ребра равна *len*, тогда в виде кода релаксация выглядит как:

На этом текущая итерация заканчивается, алгоритм переходит к следующей итерации (снова выбирается вершина с наименьшей величиной *distance*, из неё производятся релаксации, и т.д.). При этом в конце концов, после *num\_vert* итераций, все вершины графа станут помеченными, и алгоритм свою работу завершает. Утверждается, что найденные значения *distance[vert]* и есть искомые длины кратчайших путей из *start* в *vert*.

Стоит заметить, что, если не все вершины графа достижимы из вершины *start*, то значения *distance[vert]* для них так и останутся бесконечными. Понятно, что несколько последних итераций алгоритма будут как раз выбирать эти вершины, но никакой полезной работы производить эти итерации не будут (поскольку бесконечное расстояние не сможет прорелаксировать другие, даже тоже бесконечные расстояния). Поэтому алгоритм можно сразу останавливать, как только в качестве выбранной вершины берётся вершина с бесконечным расстоянием.

**Восстановление путей.** Разумеется, обычно нужно знать не только длины кратчайших путей, но и получить сами пути. Покажем, как сохранить информацию, достаточную для последующего восстановления кратчайшего пути из *start* до любой вершины. Для этого достаточно так называемого **массива предков**: массива *parent[]*, в котором для каждой вершины *vert≠start* хранится номер вершины *parent[vert]*, являющейся предпоследней в кратчайшем пути до вершины *vert*. Здесь используется тот факт, что если мы возьмём кратчайший путь до какой-то вершины *vert*, а затем удалим из этого пути последнюю вершину, то получится путь, оканчивающийся некоторой вершиной *parent[vert]*, и этот путь будет кратчайшим для вершины *parent[vert]*. Итак, если мы будем обладать этим массивом предков, то кратчайший путь можно будет восстановить по нему, просто каждый раз беря предка от текущей вершины, пока мы не придём в стартовую вершину *start* — так мы получим искомый кратчайший путь, но записанный в обратном порядке. Итак, кратчайший путь *path* до вершины *start* равен:

Осталось понять, как строить этот массив предков. Однако это делается очень просто: при каждой успешной релаксации, т.е. когда из выбранной вершины *vert* происходит улучшение расстояния до некоторой вершины *to*, мы записываем, что предком вершины *to* является вершина *vert*:

# **Схема распараллеливания**

При реализации алгоритма Дейкстры на каждом этапе релаксации происходят независимо друг от друга. Будем распараллеливать именно их.

Для этого определим для каждого процесса определенный набор вершин, для которых он будет производить эту релаксацию.

Для системы из *num\_proc* процессов, процесс с номером *rank* работает с *size\_border* вершинами, начиная с вершины *left\_border* (нумерация вершин начинается с 0). Данные параметры рассчитываются по следующему правилу:

На каждой итерации алгоритма, рассматриваемая вершина *vert* известна каждому процессу. Каждый процесс проводит релаксацию внутри своего набора вершин, а затем выбирает среди своих не просмотренных вершин кандидата на следующую итерацию. После этого среди претендентов от каждого процесса выбирается вершина с минимальным значением *distance* для следующей итерации.

# **Описание программной реализации**

**Руководство пользователя**

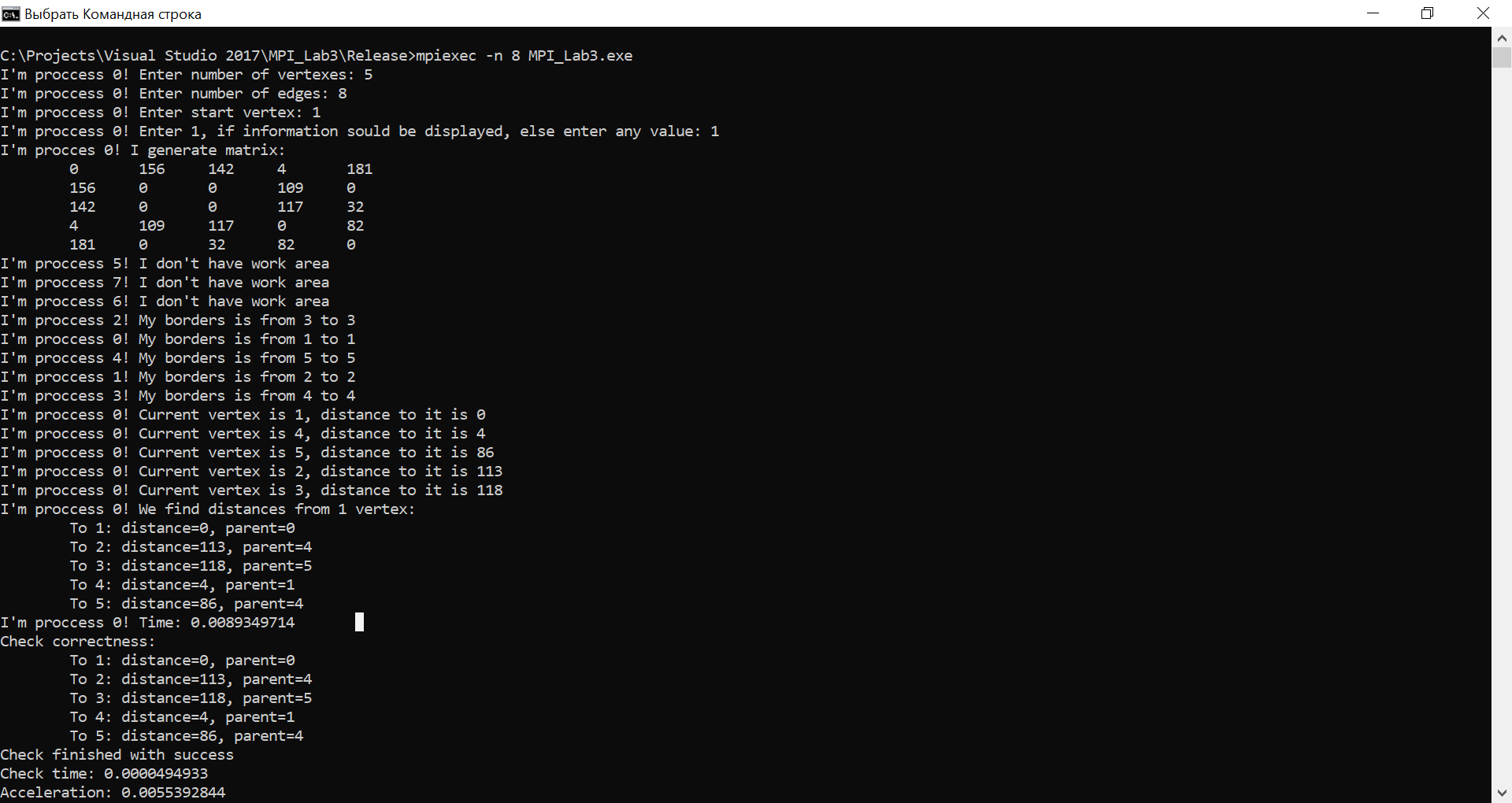
Данная программа запускается через командную строку Windows. Чтобы ее запустить необходимо указать следующую команду:

*mpiexec -n num\_proc name.exe*,

где *num\_proc –* число процессов, *name –* имя программы.

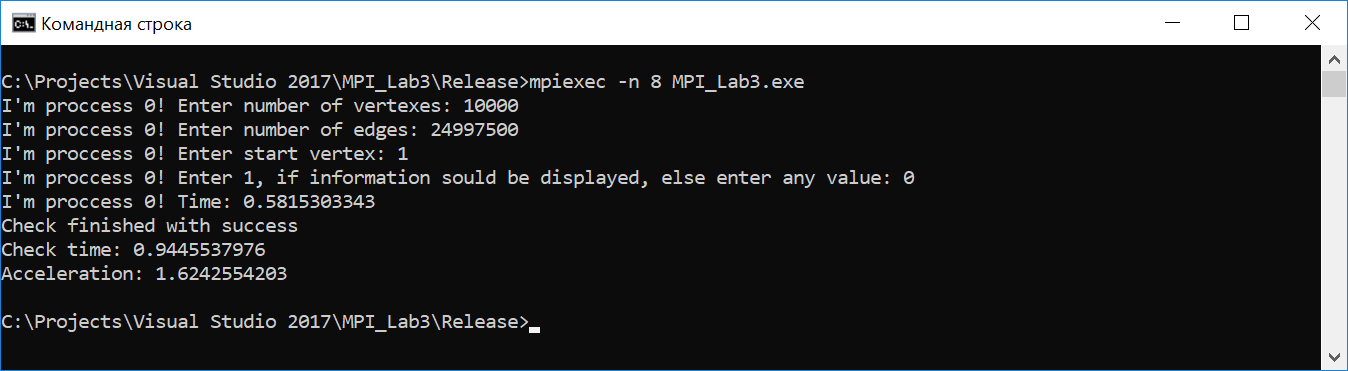
После этого программа попросит пользователя ввести число вершин, число ребер, стартовую вершину и необходимость вывода информации.

Если пользователь попросит выводить информацию, то программа в ходе своей работы будет выводить на консоль промежуточные результаты (сгенерированную матрицу, распределение вершин по процессам и т.п.).



Дополнительный вывод программы

В противном случае программа выведет только время последовательной и параллельной частей алгоритма, результаты проверки и ускорение.



Стандартный вывод программы

**Руководство программиста**

В данной программе используются следующие функции:

1. EnterInfo

Осуществляет запрос размеров графа, стартовой вершины и дополнительного вывода.

1. GenerateMatrix

Генерирует случайный взвешенный неориентированный граф с заданными пользователем параметрами.

1. Transfer

Осуществляет транспортировку частей графа между процессами.

1. DijkstraAlgorithm

Выполняет распараллеленный алгоритм поиска минимального пути.

1. FindResults

Осуществляет сбор данных со всех процессов на одном для получения общего результата.

1. Check

Делает последовательный алгоритм Дейстктры, сравнивает результаты последовательной и параллельной работ программы и выводит ускорение.

1. Main

Содержит основную реализацию программы, и вызывает другие функции в определенном порядке.

# **Подтверждение корректности**

В программе осущетваляется проверка параллельной и последовательной частей алгоритма с выводом результата проверки на консоль. Предполагается, что правильность оптимального пути проверяется путем сравнения его длины. Алгоритм не предполагает никаких расхождений в длинах путей (только в найденых предках вершин, ибо оптимальный путей может быть несколько).

# **Результаты экспериментов**

Ниже представлены таблицы и графики времени работы алгоритма а также ускорения при разных входных параметрах и числе процессов.

**Результаты для времени работы программы**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Входные денные | 1 процесс | 2 процесса | 4 процесса | 6 процессов | 8 процессов |
| (10, 23) | 6,99734E-05 | 0,002021263 | 0,005324802 | 0,00619463 | 0,024522531 |
| (100, 2475) | 0,000207076 | 0,002935468 | 0,006504677 | 0,00807026 | 0,036778677 |
| (1000, 249750) | 0,022514922 | 0,017233925 | 0,023569642 | 0,02697444 | 0,0552835 |
| (10000, 24997500) | 1,768365445 | 1,031301982 | 0,940192694 | 0,75448683 | 0,4513378 |
| (20000, 99995000) | 6,169073826 | 5,107451764 | 3,257112251 | 2,92632489 | 2,305558843 |

**Ускорение программы**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Входные денные | 1 процесс | 2 процесса | 4 процесса | 6 процессов | 8 процессов |
| (10, 23) | 0,048780376 | 0,001688717 | 0,000961539 | 0,000826523 | 0,000208788 |
| (100, 2475) | 0,659340661 | 0,050581396 | 0,034983385 | 0,028619767 | 0,006156226 |
| (1000, 249750) | 0,704644234 | 1,111309171 | 0,652747942 | 0,568470558 | 0,459280907 |
| (10000, 24997500) | 0,737951084 | 1,213475578 | 1,72193356 | 1,933150411 | 2,248777552 |
| (20000, 99995000) | 0,757763763 | 1,259091328 | 1,963002969 | 2,265991629 | 2,56701569 |

# **Вывод**

Распараллеливание программы может значительно улучшить время ее работы при больших размерах данных. Однако стоит обратить внимание на издержки, которые такой подход за собой несет. В некоторых ситуациях лучше использовать последовательную реализацию.

Кроме того, число процессов также нужно подбирать в соответствии с входными данными. Ибо от увеличения их количества увеличивается эти самые издержки.