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

Отчет по лабораторной работе

«Поиск кратчайших путей из одной вершины (алгоритм Дейкстры)»

Выполнила:
студентка группы 381906-1
Тырина А. К.

Проверил:
доцент кафедры МОСТ,
кандидат технических наук
Сысоев А. В.

Нижний Новгород
2022

Оглавление

Введение

Задача о кратчайшем пути — задача поиска самого короткого пути (цепи) между двумя точками (вершинами) на графе, в которой минимизируется сумма весов рёбер, составляющих путь.

Значимость данной задачи определяется её различными практическими применениями. Например, в GPS-навигаторах осуществляется поиск кратчайшего пути между точкой отправления и точкой назначения. В качестве вершин выступают перекрёстки, а дороги являются рёбрами, которые лежат между ними. Если сумма длин дорог между перекрёстками минимальна, тогда найденный путь самый короткий.

Задача о кратчайшем пути является одной из важнейших классических задач теории графов. Сегодня известно множество алгоритмов для её решения. Один из таких алгоритмов - алгоритм Дейкстры, изобретённый нидерландским учёным Эдсгером Дейкстрой в 1959 году. Алгоритм работает только для графов без рёбер отрицательного веса.

Постановка задачи

В данном лабораторной работе требуется реализовать последовательную версию и параллельные версии алгоритма Дейкстры, провести вычислительные эксперименты для сравнения времени работы алгоритмов, используя при этом фрэймворк для разработки автоматических тестов Google Test, сделать выводы об эффективности реализованных алгоритмов.

Параллельные алгоритмы должны быть реализованы при помощи библиотек параллельного программирования OpenMP и TBB и встроенных в C++ средств создания потоков std::thread.

Описание алгоритма

Алгоритм Дейкстры выглядит следующим образом:

1. Создать множество sptSet (множество дерева наикратчайших путей), которое будет отслеживать вершины, включенные в дерево наикратчайших путей, то есть чьи минимальные расстояния от источника уже посчитаны. Изначально множество пустое.
2. Присвоить значение расстояния всем вершинам во входном графе. Инициализировать все значения расстояний как бесконечность. Присвоить значению расстояния вершины-источника 0, так как мы выбираем её первой.
3. Пока в sptSet не включены все вершины:
   (a) Выбрать вершину u, которой еще нет в sptSet и у которой значение расстояния минимально.
   (b) Включить u в sptSet.
   (c) Обновить значения расстояния у всех вершин, смежных u. Чтобы обновить значения расстояний, проитерироваться по всем смежным вершинам. Для каждой смежной вершины v, если сумма значения расстояния u (от источника) и веса ребра u-v меньше, чем значение расстояния v, то обновить значение расстояния v.

Описание схемы распараллеливания

В нашей программе есть два цикла, которые проходят по всем вершинам - поиск вершины с минимальным расстоянием и обновление смежных вершин. Эти циклы можно распараллелить. Под этим имеется в виду выполнение каждым потоком своей части всего цикла. В OpenMP для этого используется директива parallel for, в TBB - функции parallel_reduce и parallel_for. Создание потоков и распределение частей циклов в них происходит автоматически. В std::thread мы сами создаем потоки и вручную прописываем, какой поток выполняет какую часть цикла.

Описание программной реализации

Программа состоит из заголовочного файла dijkstra.h и двух файлов исходного кода dijkstra.cpp и main.cpp.

В заголовочном файле находятся прототипы функций для последовательного и параллельных алгоритмов Дейкстры.

Определения сокращений для типов данных, которые мы используем в программе:

```
using VectorInt = std::vector<int>;
using VectorBool = std::vector<bool>;
using Graph = std::vector<std::vector<int>>;
```

Функция для создания случайного графа:

```
Graph getRandomGraph(int V);
```

Входной параметр - число вершин графа.

Последовательная функция алгоритма Дейкстры для нахождения кратчайших путей для одной вершины графа:

```
VectorInt dijkstra(const Graph& graph, int src, int V);
```

Функция принимает на вход граф, индекс вершины-источника и количество вершин в графе.

Параллельная функция алгоритма Дейкстры для нахождения кратчайших путей для одной вершины графа:

```
VectorInt dijkstra_parallel(const Graph& graph, int src, int V);
```

Параметры совпадают с последовательной функцией.

Последовательная функция прохода алгоритмом Дейкстры по всем вершинам заданного графа:

```
Graph sequentialDijkstra(const Graph& graph, int V);
```

Функция принимает на вход граф и количество его вершин.

Параллельная функция прохода алгоритмом Дейкстры по всем вершинам заданного графа (функция совпадает для всех библиотек):

```
Graph parallelDijkstra(const Graph& graph, int V);
```

Параметры совпадают с последовательной функцией.

В файле исходного кода dijkstra.cpp содержится реализация функций, объявленных в заголовочном файле. В файле исходного кода main.cpp содержатся тесты для проверки корректности программы.

Подтверждение корректности

Для подтверждения корректности работы своей программы я написала на фрэймфорке Google Test 5 тестов для каждой версии алгоритма. В каждом тесте я создаю случайный граф, вычисляю результаты работы последовательного и параллельного алгоритмов и сравниваю их, они должны совпадать.

Успешное прохождение всех тестов подтверждает корректность работы программы.

Результаты экспериментов

Для проведения эксперименотов по вычислению эффективности работы разных реализаций программы использовалась система со следующей конфигурацией:

- Процессор: Intel Core i5-10210U, 1.6 ГГц, ядер: 4, потоков: 8;
- Оперативная память: 8 ГБ (DDR4), 2666 МГц;
- Операционная система: Windows 10 Home.

Эксперименты проводились на 8 потоках. Число вершин в тестовом графе - 20000.

Результаты экспериментов представлены в Таблице 1.

Версия	Время работы (сек.)	Ускорение
Последовательный	1.7573	-
OpenMP	0.84735	2.07
TBB	1.29115	1.37
std::thread	1.46301	1.21

Таблица 1: Результаты экспериментов с графом на 20000 вершин

Выводы из результатов экспериментов

Из данных, полученных в результате экспериментов (см. Таблицу 1), можно сделать вывод, что лишь версия с OpenMP позволяет достичь значимого ускорения. Версии с TBB и std::thread дают лишь незначительное ускорение.

Это можно связать с тем, что алгоритм Дейкстры заведомо плохо подходит для распараллеливания. В итерациях циклов алгоритма не проводится никаких затратных вычислений, происходит лишь сравнение. Таким образом, даже на больших числах алгоритм работает быстро. Тем временем, задержки на создание потоков, распределение данных и синхронизацию значительно ударяют по общей эффективности. Сильнее всего от этого страдает реализация с std::thread, так как там вся работа с потоками ведется вручную.

Тем не менее, можно сказать, что больше всего для создания параллельной версии алгоритма Дейкстры подходит библиотека OpenMP.

Заключение

Таким образом, в рамках данной лабораторной работы были разработаны последовательный и три версии параллельного алгоритма Дейкстры. Проведенные тесты показали корректность реализованной программы, а проведенные эксперименты доказали сложности с реализацией параллельного алгоритма Дейкстры.

Литература

1. Wikipedia - Электронный ресурс. URL:
   https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm
2. Geeksforgeeks - Электронный ресурс. URL:
   https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/
3. Educative - Электронный ресурс. URL:
   https://www.educative.io/blog/modern-multithreading-and-concurrency-in-cpp
4. А.В. Сысоев, И.Б. Мееров, А.А. Сиднев «Средства разработки параллельных программ для систем с общей памятью. Библиотека Intel Threading Building Blocks». Нижний Новгород, 2007, 128 с.
5. А.В. Сысоев, И.Б. Мееров, А.Н. Свистунов, А.Л. Курылев, А.В. Сенин, А.В. Шишков, К.В. Корняков, А.А. Сиднев «Параллельное программирование в системах с общей памятью. Инструментальная поддержка». Нижний Новгород, 2007, 110 с.

Приложение

Последовательный алгоритм Дейкстры

seq/dijkstra.h

```
// Copyright 2022 Tyrina Anastasia
#ifndef MODULES_TASK_1_TYRINA_A_DIJKSTRA_DIJKSTRA_H_
#define MODULES_TASK_1_TYRINA_A_DIJKSTRA_DIJKSTRA_H_

#include <vector>

using VectorInt = std::vector<int>;
using VectorBool = std::vector<bool>;
using Graph = std::vector<std::vector<int>>;

Graph getRandomGraph(int V);
Graph sequentialDijkstra(const Graph& graph, int V);

#endif  // MODULES_TASK_1_TYRINA_A_DIJKSTRA_DIJKSTRA_H_
```

seq/dijkstra.cpp

```
// Copyright 2022 Tyrina Anastasia
#include "../../../modules/task_1/tyrina_a_dijkstra/dijkstra.h"

#include <climits>
#include <random>
#include <vector>

Graph getRandomGraph(int V) {
  Graph graph(V, VectorInt(V));
  std::random_device dev;
  std::mt19937 gen(dev());

  for (int i = 0; i < V; ++i) {
    for (int j = i + 1; j < V; ++j) {
      graph[i][j] = gen() % 20;
      graph[j][i] = graph[i][j];
    }
    graph[i][i] = 0;
  }
  return graph;
}

int minDistance(const VectorInt& dist, const VectorBool& sptSet, int V) {
  int min = INT_MAX, min_index = 0;

  for (int v = 0; v < V; v++)
    if (sptSet[v] == false && dist[v] <= min) min = dist[v], min_index = v;

  return min_index;
}

VectorInt dijkstra(const Graph& graph, int src, int V) {
  VectorInt dist(V);
  VectorBool visited(V);

  for (int i = 0; i < V; i++) {
    dist[i] = INT_MAX;
```

```
    visited[i] = false;
  }

  dist[src] = 0;

  for (int count = 0; count < V - 1; count++) {
    int u = minDistance(dist, visited, V);

    visited[u] = true;

    for (int v = 0; v < V; v++) {
      if (!visited[v] && graph[u][v] && dist[u] != INT_MAX &&
          dist[u] + graph[u][v] < dist[v]) {
        dist[v] = dist[u] + graph[u][v];
      }
    }
  }

  return dist;
}

Graph sequentialDijkstra(const Graph& graph, int V) {
  Graph result(V, VectorInt(V));

  for (int src = 0; src < V; ++src) {
    VectorInt temp = dijkstra(graph, src, V);
    for (int i = 0; i < V; ++i) {
      result[src][i] = temp[i];
    }
  }

  return result;
}
```

seq/main.cpp

```
// Copyright 2022 Tyrina Anastasia
#include <gtest/gtest.h>

#include "./dijkstra.h"

TEST(DIJKSTRA_SEQ, TEST_1) {
  int V = 9;
  Graph graph = {{0, 4, 0, 0, 0, 0, 0, 8, 0},  {4, 0, 8, 0, 0, 0, 0, 11, 0},
                 {0, 8, 0, 7, 0, 4, 0, 0, 2},  {0, 0, 7, 0, 9, 14, 0, 0, 0},
                 {0, 0, 0, 9, 0, 10, 0, 0, 0}, {0, 0, 4, 14, 10, 0, 2, 0, 0},
                 {0, 0, 0, 0, 0, 2, 0, 1, 6},  {8, 11, 0, 0, 0, 0, 1, 0, 7},
                 {0, 0, 2, 0, 0, 0, 6, 7, 0}};

  Graph result = sequentialDijkstra(graph, V);
}

TEST(DIJKSTRA_SEQ, TEST_2) {
  int V = 10;
  Graph graph = getRandomGraph(V);
  Graph result = sequentialDijkstra(graph, V);
}

TEST(DIJKSTRA_SEQ, TEST_3) {
  int V = 20;
  Graph graph = getRandomGraph(V);
```

```
  Graph result = sequentialDijkstra(graph, V);
}

TEST(DIJKSTRA_SEQ, TEST_4) {
  int V = 30;
  Graph graph = getRandomGraph(V);
  Graph result = sequentialDijkstra(graph, V);
}

TEST(DIJKSTRA_SEQ, TEST_5) {
  int V = 40;
  Graph graph = getRandomGraph(V);
  Graph result = sequentialDijkstra(graph, V);
}
```

OpenMP

omp/dijkstra.h

```
// Copyright 2022 Tyrina Anastasia
#ifndef MODULES_TASK_2_TYRINA_A_DIJKSTRA_DIJKSTRA_H_
#define MODULES_TASK_2_TYRINA_A_DIJKSTRA_DIJKSTRA_H_

#include <vector>

using VectorInt = std::vector<int>;
using VectorBool = std::vector<bool>;
using Graph = std::vector<std::vector<int>>;

Graph getRandomGraph(int V);
Graph sequentialDijkstra(const Graph& graph, int V);
Graph parallelDijkstra(const Graph& graph, int V);

VectorInt dijkstra(const Graph& graph, int src, int V);
VectorInt dijkstra_parallel(const Graph& graph, int src, int V);

#endif  // MODULES_TASK_2_TYRINA_A_DIJKSTRA_DIJKSTRA_H_
```

omp/dijkstra.cpp

```
// Copyright 2022 Tyrina Anastasia
#include "../../../modules/task_2/tyrina_a_dijkstra/dijkstra.h"

#include <omp.h>

#include <climits>
#include <random>
#include <vector>

Graph getRandomGraph(int V) {
  Graph graph(V, VectorInt(V));
  std::random_device dev;
  std::mt19937 gen(dev());

  for (int i = 0; i < V; ++i) {
    for (int j = i + 1; j < V; ++j) {
      graph[i][j] = gen() % 20;
      graph[j][i] = graph[i][j];
    }
    graph[i][i] = 0;
```

```
  }
  return graph;
}

int minDistance(const VectorInt& dist, const VectorBool& sptSet, int V) {
  int min = INT_MAX, min_index = 0;

  for (int v = 0; v < V; v++)
    if (sptSet[v] == false && dist[v] <= min) min = dist[v], min_index = v;

  return min_index;
}

VectorInt dijkstra(const Graph& graph, int src, int V) {
  VectorInt dist(V);
  VectorBool visited(V);

  for (int i = 0; i < V; i++) {
    dist[i] = INT_MAX;
    visited[i] = false;
  }

  dist[src] = 0;

  for (int count = 0; count < V - 1; count++) {
    int u = minDistance(dist, visited, V);

    visited[u] = true;

    for (int v = 0; v < V; v++) {
      if (!visited[v] && graph[u][v] && dist[u] != INT_MAX &&
          dist[u] + graph[u][v] < dist[v]) {
        dist[v] = dist[u] + graph[u][v];
      }
    }
  }

  return dist;
}

VectorInt dijkstra_parallel(const Graph& graph, int src, int V) {
  VectorInt dist(V, INT_MAX);
  VectorBool visited(V, false);

  dist[src] = 0;

  for (int count = 0; count < V - 1; count++) {
    int min = INT_MAX, u = 0;
#pragma omp parallel
    {
      int local_min = INT8_MAX;
      int local_ind = 0;
#pragma omp for
      for (int i = 0; i < V; ++i)
        if (!visited[i] && dist[i] <= local_min) {
          local_min = dist[i];
          local_ind = i;
        }
#pragma omp critical
      if (local_min < min) {
```

```
        min = local_min;
        u = local_ind;
      }

#pragma omp barrier

#pragma omp single
      { visited[u] = true; }

#pragma omp for
      for (int v = 0; v < V; v++) {
        if (!visited[v] && graph[u][v] && dist[u] != INT_MAX &&
            dist[u] + graph[u][v] < dist[v]) {
          dist[v] = dist[u] + graph[u][v];
        }
      }
    }
  }
  return dist;
}

Graph sequentialDijkstra(const Graph& graph, int V) {
  Graph result(V, VectorInt(V));

  for (int src = 0; src < V; ++src) {
    result[src] = dijkstra(graph, src, V);
  }

  return result;
}

Graph parallelDijkstra(const Graph& graph, int V) {
  Graph result(V, VectorInt(V));

  for (int src = 0; src < V; ++src) {
    result[src] = dijkstra_parallel(graph, src, V);
  }

  return result;
}
```

omp/main.cpp

```
// Copyright 2022 Tyrina Anastasia
#include <gtest/gtest.h>

#include "./dijkstra.h"

#include <omp.h>

TEST(DIJKSTRA_OMP, TEST_1) {
  int V = 9;
  Graph graph = {{0, 4, 0, 0, 0, 0, 0, 8, 0},  {4, 0, 8, 0, 0, 0, 0, 11, 0},
                 {0, 8, 0, 7, 0, 4, 0, 0, 2},  {0, 0, 7, 0, 9, 14, 0, 0, 0},
                 {0, 0, 0, 9, 0, 10, 0, 0, 0}, {0, 0, 4, 14, 10, 0, 2, 0, 0},
                 {0, 0, 0, 0, 0, 2, 0, 1, 6},  {8, 11, 0, 0, 0, 0, 1, 0, 7},
                 {0, 0, 2, 0, 0, 0, 6, 7, 0}};

  Graph result_sequential = sequentialDijkstra(graph, V);
  Graph result_parallel = parallelDijkstra(graph, V);
```

```
  for (int i = 0; i < V; i++) {
    ASSERT_EQ(result_sequential[i], result_parallel[i]);
  }
}

TEST(DIJKSTRA_OMP, TEST_2) {
  int V = 10;
  Graph graph = getRandomGraph(V);
  Graph result_sequential = sequentialDijkstra(graph, V);
  Graph result_parallel = parallelDijkstra(graph, V);

  for (int i = 0; i < V; i++) {
    ASSERT_EQ(result_sequential[i], result_parallel[i]);
  }
}

TEST(DIJKSTRA_OMP, TEST_3) {
  int V = 20;
  Graph graph = getRandomGraph(V);
  Graph result_sequential = sequentialDijkstra(graph, V);
  Graph result_parallel = parallelDijkstra(graph, V);

  for (int i = 0; i < V; i++) {
    ASSERT_EQ(result_sequential[i], result_parallel[i]);
  }
}

TEST(DIJKSTRA_OMP, TEST_4) {
  int V = 30;
  Graph graph = getRandomGraph(V);
  Graph result_sequential = sequentialDijkstra(graph, V);
  Graph result_parallel = parallelDijkstra(graph, V);

  for (int i = 0; i < V; i++) {
    ASSERT_EQ(result_sequential[i], result_parallel[i]);
  }
}

int V = 20000;

TEST(DIJKSTRA_OMP, TEST_TIME) {
  Graph graph = getRandomGraph(V);

  double start = omp_get_wtime();
  dijkstra_parallel(graph, 0, V);
  double end = omp_get_wtime();
  double ptime = end - start;
  std::cout << "\tparallel time: " << ptime << "\n";

  start = omp_get_wtime();
  dijkstra(graph, 0, V);
  end = omp_get_wtime();
  double stime = end - start;
  std::cout << "\tsequential time: " << stime << "\n";

  std::cout << "\tefficiency: " << stime / ptime << "\n";
}
```

TBB

tbb/dijkstra.h

```
// Copyright 2022 Tyrina Anastasia
#ifndef MODULES_TASK_3_TYRINA_A_DIJKSTRA_DIJKSTRA_H_
#define MODULES_TASK_3_TYRINA_A_DIJKSTRA_DIJKSTRA_H_

#include <vector>

using VectorInt = std::vector<int>;
using VectorBool = std::vector<bool>;
using Graph = std::vector<std::vector<int>>;

Graph getRandomGraph(int V);
Graph sequentialDijkstra(const Graph& graph, int V);
Graph parallelDijkstra(const Graph& graph, int V);

VectorInt dijkstra(const Graph& graph, int src, int V);
VectorInt dijkstra_parallel(const Graph& graph, int src, int V);

#endif  // MODULES_TASK_3_TYRINA_A_DIJKSTRA_DIJKSTRA_H_
```

tbb/dijkstra.cpp

```
// Copyright 2022 Tyrina Anastasia
#include "../../../modules/task_3/tyrina_a_dijkstra/dijkstra.h"

#include <omp.h>
#include <tbb/tbb.h>

#include <climits>
#include <iostream>
#include <random>
#include <vector>

Graph getRandomGraph(int V) {
  Graph graph(V, VectorInt(V));
  std::random_device dev;
  std::mt19937 gen(dev());

  for (int i = 0; i < V; ++i) {
    for (int j = i + 1; j < V; ++j) {
      graph[i][j] = gen() % 20;
      graph[j][i] = graph[i][j];
    }
    graph[i][i] = 0;
  }
  return graph;
}

int minDistance(const VectorInt& dist, const VectorBool& sptSet, int V) {
  int min = INT_MAX, min_index = 0;

  for (int v = 0; v < V; v++)
    if (sptSet[v] == false && dist[v] <= min) min = dist[v], min_index = v;

  return min_index;
}

VectorInt dijkstra(const Graph& graph, int src, int V) {
  VectorInt dist(V);
  VectorBool visited(V);
```

```
  for (int i = 0; i < V; i++) {
    dist[i] = INT_MAX;
    visited[i] = false;
  }

  dist[src] = 0;

  for (int count = 0; count < V - 1; count++) {
    int u = minDistance(dist, visited, V);

    visited[u] = true;

    for (int v = 0; v < V; v++) {
      if (!visited[v] && graph[u][v] && dist[u] != INT_MAX &&
          dist[u] + graph[u][v] < dist[v]) {
        dist[v] = dist[u] + graph[u][v];
      }
    }
  }

  return dist;
}

VectorInt dijkstra_parallel(const Graph& graph, int src, int V) {
  VectorInt dist(V, INT_MAX);
  VectorBool visited(V, false);

  dist[src] = 0;

  struct vertex {
    int min;
    int min_index;
  };
  vertex current = {INT8_MAX, 0};

  for (int count = 0; count < V - 1; count++) {
    current.min = INT8_MAX;
    current = tbb::parallel_reduce(
        tbb::blocked_range<int>(0, V), current,
        [&](const tbb::blocked_range<int>& range, vertex cur) -> vertex {
          for (int i = range.begin(); i != range.end(); ++i) {
            if (!visited[i] && dist[i] <= cur.min) {
              cur.min = dist[i];
              cur.min_index = i;
            }
          }
          return cur;
        },
        [](vertex a, vertex b) { return a.min < b.min ? a : b; });

    int u = current.min_index;
    visited[u] = true;

    tbb::parallel_for(
        tbb::blocked_range<int>(0, V), [&](tbb::blocked_range<int> r) {
          for (int v = r.begin(); v < r.end(); ++v) {
            if (!visited[v] && graph[u][v] && dist[u] != INT_MAX &&
                dist[u] + graph[u][v] < dist[v]) {
              dist[v] = dist[u] + graph[u][v];
            }
```

```
        }
      });
  }

  return dist;
}

Graph sequentialDijkstra(const Graph& graph, int V) {
  Graph result(V, VectorInt(V));

  for (int src = 0; src < V; ++src) {
    result[src] = dijkstra(graph, src, V);
  }

  return result;
}

Graph parallelDijkstra(const Graph& graph, int V) {
  Graph result(V, VectorInt(V));

  for (int src = 0; src < V; ++src) {
    result[src] = dijkstra_parallel(graph, src, V);
  }

  return result;
}
```

tbb/main.cpp

```
// Copyright 2022 Tyrina Anastasia
#include <gtest/gtest.h>
#include <omp.h>

#include "./dijkstra.h"

TEST(DIJKSTRA_TBB, TEST_1) {
  int V = 9;
  Graph graph = {{0, 4, 0, 0, 0, 0, 0, 8, 0},  {4, 0, 8, 0, 0, 0, 0, 11, 0},
                 {0, 8, 0, 7, 0, 4, 0, 0, 2},  {0, 0, 7, 0, 9, 14, 0, 0, 0},
                 {0, 0, 0, 9, 0, 10, 0, 0, 0}, {0, 0, 4, 14, 10, 0, 2, 0, 0},
                 {0, 0, 0, 0, 0, 2, 0, 1, 6},  {8, 11, 0, 0, 0, 0, 1, 0, 7},
                 {0, 0, 2, 0, 0, 0, 6, 7, 0}};

  Graph result_sequential = sequentialDijkstra(graph, V);
  Graph result_parallel = parallelDijkstra(graph, V);

  for (int i = 0; i < V; i++) {
    ASSERT_EQ(result_sequential[i], result_parallel[i]);
  }
}

TEST(DIJKSTRA_TBB, TEST_2) {
  int V = 4;
  Graph graph = getRandomGraph(V);
  Graph result_sequential = sequentialDijkstra(graph, V);
  Graph result_parallel = parallelDijkstra(graph, V);

  for (int i = 0; i < V; i++) {
    ASSERT_EQ(result_sequential[i], result_parallel[i]);
  }
}
```

```
TEST(DIJKSTRA_TBB, TEST_3) {
  int V = 6;
  Graph graph = getRandomGraph(V);
  Graph result_sequential = sequentialDijkstra(graph, V);
  Graph result_parallel = parallelDijkstra(graph, V);

  for (int i = 0; i < V; i++) {
    ASSERT_EQ(result_sequential[i], result_parallel[i]);
  }
}

TEST(DIJKSTRA_TBB, TEST_4) {
  int V = 8;
  Graph graph = getRandomGraph(V);
  Graph result_sequential = sequentialDijkstra(graph, V);
  Graph result_parallel = parallelDijkstra(graph, V);

  for (int i = 0; i < V; i++) {
    ASSERT_EQ(result_sequential[i], result_parallel[i]);
  }
}

int V = 20000;

TEST(DIJKSTRA_TBB, TEST_TIME) {
  Graph graph = getRandomGraph(V);

  double start = omp_get_wtime();
  dijkstra(graph, 0, V);
  double end = omp_get_wtime();
  double stime = end - start;
  std::cout << "\tsequential time: " << stime << "\n";

  start = omp_get_wtime();
  dijkstra_parallel(graph, 0, V);
  end = omp_get_wtime();
  double ptime = end - start;
  std::cout << "\tparallel time: " << ptime << "\n";

  std::cout << "\tefficiency: " << stime / ptime << "\n";
}
```

std::threads

std/dijkstra.h

```
// Copyright 2022 Tyrina Anastasia
#ifndef MODULES_TASK_4_TYRINA_A_DIJKSTRA_DIJKSTRA_H_
#define MODULES_TASK_4_TYRINA_A_DIJKSTRA_DIJKSTRA_H_

#include <vector>

using VectorInt = std::vector<int>;
using VectorBool = std::vector<bool>;
using Graph = std::vector<std::vector<int>>;

Graph getRandomGraph(int V);
Graph sequentialDijkstra(const Graph& graph, int V);
Graph parallelDijkstra(const Graph& graph, int V);
```

```
VectorInt dijkstra(const Graph& graph, int src, int V);
VectorInt dijkstra_parallel(const Graph& graph, int src, int V);

#endif  // MODULES_TASK_4_TYRINA_A_DIJKSTRA_DIJKSTRA_H_
```

std/dijkstra.cpp

```
// Copyright 2022 Tyrina Anastasia
#include "../../../modules/task_4/tyrina_a_dijkstra/dijkstra.h"

#include <omp.h>

#include <climits>
#include <random>
#include <vector>

#include "../../../3rdparty/unapproved/unapproved.h"

Graph getRandomGraph(int V) {
  Graph graph(V, VectorInt(V));
  std::random_device dev;
  std::mt19937 gen(dev());

  for (int i = 0; i < V; ++i) {
    for (int j = i + 1; j < V; ++j) {
      graph[i][j] = gen() % 20;
      graph[j][i] = graph[i][j];
    }
    graph[i][i] = 0;
  }
  return graph;
}

int minDistance(const VectorInt& dist, const VectorBool& sptSet, int V) {
  int min = INT_MAX, min_index = 0;

  for (int v = 0; v < V; v++)
    if (sptSet[v] == false && dist[v] <= min) min = dist[v], min_index = v;

  return min_index;
}

int minDistanceParallel(const VectorInt& dist, const VectorBool& sptSet, int V)
   {
  int min1 = INT_MAX, min_index1 = 0;
  int min2 = INT_MAX, min_index2 = 0;

  std::thread thread_1([V, &dist, &sptSet, &min1, &min_index1]() {
    for (int v = 0; v < V / 2; v++)
      if (sptSet[v] == false && dist[v] <= min1) min1 = dist[v], min_index1 = v;
    });

  std::thread thread_2([V, &dist, &sptSet, &min2, &min_index2]() {
    for (int v = V / 2; v < V; v++)
      if (sptSet[v] == false && dist[v] <= min2) min2 = dist[v], min_index2 = v;
    });

  thread_1.join();
  thread_2.join();
```

```
  int min_index;

  if (min1 < min2) {
    min_index = min_index1;
  }
  else {
    min_index = min_index2;
  }

  return min_index;
}

VectorInt dijkstra(const Graph& graph, int src, int V) {
  VectorInt dist(V);
  VectorBool visited(V);

  for (int i = 0; i < V; i++) {
    dist[i] = INT_MAX;
    visited[i] = false;
  }

  dist[src] = 0;

  for (int count = 0; count < V - 1; count++) {
    int u = minDistance(dist, visited, V);

    visited[u] = true;

    for (int v = 0; v < V; v++) {
      if (!visited[v] && graph[u][v] && dist[u] != INT_MAX &&
        dist[u] + graph[u][v] < dist[v]) {
        dist[v] = dist[u] + graph[u][v];
      }
    }
  }

  return dist;
}

VectorInt dijkstra_parallel(const Graph& graph, int src, int V) {
  VectorInt dist(V, INT_MAX);
  VectorBool visited(V, false);

  dist[src] = 0;

  for (int count = 0; count < V - 1; count++) {
    int u = minDistanceParallel(dist, visited, V);

    visited[u] = true;

    for (int v = 0; v < V; v++) {
      if (!visited[v] && graph[u][v] && dist[u] != INT_MAX &&
        dist[u] + graph[u][v] < dist[v]) {
        dist[v] = dist[u] + graph[u][v];
      }
    }
  }
  return dist;
}
```

```
Graph sequentialDijkstra(const Graph& graph, int V) {
  Graph result(V, VectorInt(V));

  for (int src = 0; src < V; ++src) {
    result[src] = dijkstra(graph, src, V);
  }

  return result;
}

Graph parallelDijkstra(const Graph& graph, int V) {
  Graph result(V, VectorInt(V));

  for (int src = 0; src < V; ++src) {
    result[src] = dijkstra_parallel(graph, src, V);
  }

  return result;
}
```

std/main.cpp

```
// Copyright 2022 Tyrina Anastasia
#include <gtest/gtest.h>

#include "../../../3rdparty/unapproved/unapproved.h"
#include "./dijkstra.h"

TEST(DIJKSTRA_STD, TEST_1) {
  int V = 4;
  Graph graph = getRandomGraph(V);
  Graph result_sequential = sequentialDijkstra(graph, V);
  Graph result_parallel = parallelDijkstra(graph, V);
  for (int i = 0; i < V; i++) {
    ASSERT_EQ(result_sequential[i], result_parallel[i]);
  }
}

TEST(DIJKSTRA_STD, TEST_2) {
  int V = 4;
  Graph graph = getRandomGraph(V);
  Graph result_sequential = sequentialDijkstra(graph, V);
  Graph result_parallel = parallelDijkstra(graph, V);
}

TEST(DIJKSTRA_STD, TEST_3) {
  int V = 4;
  Graph graph = getRandomGraph(V);
  Graph result_sequential = sequentialDijkstra(graph, V);
  Graph result_parallel = parallelDijkstra(graph, V);
}

TEST(DIJKSTRA_STD, TEST_4) {
  int V = 4;
  Graph graph = getRandomGraph(V);
  Graph result_sequential = sequentialDijkstra(graph, V);
  Graph result_parallel = parallelDijkstra(graph, V);
}

int V = 20000;
```

```
TEST(DIJKSTRA_STD, TEST_TIME) {
  Graph graph = getRandomGraph(V);

  auto t1 = clock();
  dijkstra_parallel(graph, 0, V);
  auto t2 = clock();

  auto t3 = clock();
  dijkstra(graph, 0, V);
  auto t4 = clock();

  printf("sequential time: %f\n", static_cast<float>(t4 - t3) / CLOCKS_PER_SEC);
  printf("parallel time: %f\n", static_cast<float>(t2 - t1) / CLOCKS_PER_SEC);
  printf("eff: %f\n", (static_cast<float>(t4 - t3) / CLOCKS_PER_SEC) /
    (static_cast<float>(t2 - t1) / CLOCKS_PER_SEC));
}
```