МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«Вятский государственный университет»
Факультет автоматики и вычислительной техники
Кафедра электронных вычислительных машин

Лабораторная работа № 1 по курсу «Параллельное программирование»

Выполнил студент группы ИВТ-32 _____/Рзаев А. Э./
Проверил доцент кафедры ЭВМ _____/Чистяков Г. А./

Киров 2017

1 Задание

Необходимо реализовать «жадный» алгоритм раскраски графа.

1. Изучить алгоритм
2. Провести доказательную оценку алгоритма по временной сложности и затратами по памяти
3. Реализовать алгоритм с помощью языка C++
4. Построить набор тестовых примеров и провести оценку эффективности реализованного алгоритма

2 Изучение предметной области

Жадный алгоритм упорядочивает вершины $v_1 \dots v_n$ в соответствии с некоторой перестановкой и последовательно присваивает вершине v_i наименьший доступный цвет, не использовавшийся для окраски соседей v_i среди $v_1 \dots v_{i-1}$, либо добавляет новый. Качество полученной раскраски зависит от выбранного порядка. Для улучшения результата можно использовать несколько различных перестановок вершин.

Асимптотическая сложность данного алгоритма: $O(kn^2 \log n)$. Для каждой перестановки вершин (всего k перестановок), сгенерированной случайно, внешний цикл перебирает все вершины v_i и для каждой определяет цвета ее соседей, что в худшем случае будет выполнено за n шагов. Для хранения использованных цветов применяется красно-черное дерево.

3 Программная реализация

Листинг программной реализации алгоритма на C++ приведен в приложении А.

4 Тестирование

В ходе тестирования выполнялась раскраска графов, сгенерированных случайным образом. Количество вершин и ребер каждого графа и результаты тестирования приведены в таблице 1.

Таблица 1 – Результаты тестирования

Граф (кол-во вершин, кол-во ребер)	Минимальное кол-во цветов для раскраски	Время
75, 1000	11	115 мс
75, 2500	34	175 мс
90, 300	5	29 мс
95, 2500	21	171 мс
150, 6500	31	439 мс
170, 10000	41	713 мс
250, 14000	35	923 мс
250, 17000	42	1136 мс
300, 34000	77	2498 мс

5 Вывод

В ходе лабораторной работы был реализован «жадный» алгоритм раскраски графа. Для улучшения качества раскраски перебор вершин выполнялся несколько раз, порядок вершин определялся случайной перестановкой. В ходе тестирования было замечено, что время работы алгоритма увеличивается как при увеличении числа вершин, так и при увеличении числа ребер.

Приложение А
(обязательное)
Листинг программной реализации

algo.h

```
#include <vector>
#include <iostream>
#include <cstdint>
#include <set>
#include <algorithm>
#include <future>

using graph_t = std::vector<std::vector<size_t>>;

std::istream& operator>>(std::istream& is, graph_t& graph) {
    size_t n; is >> n; // vertexes
    size_t m; is >> m; // edges
    graph.clear();
    graph.resize(n);

    for (size_t i = 0; i < m; ++i) {
        size_t a, b; is >> a >> b;
        graph[a].push_back(b);
        graph[b].push_back(a);
    }

    return is;
}

std::ostream& operator<<(std::ostream& os, const graph_t& graph) {
    std::set<std::pair<size_t, size_t>> edges;
    for (size_t v = 0; v < graph.size(); ++v) {
        for (size_t to : graph[v]) {
            edges.emplace(std::minmax(v, to));
        }
    }
    os << graph.size() << ' ' << edges.size() << std::endl;
    for (const auto& p : edges) {
        os << p.first << ' ' << p.second << std::endl;
    }

    return os;
}

namespace improved {
    size_t _colorize(const graph_t&, const std::vector<size_t>&);

    size_t colorize(const graph_t& graph) {
        size_t orders_count = 500;
        // generate vertexes permutations
        std::vector<size_t> order(graph.size());
        for (size_t i = 0; i < graph.size(); ++i) {
            order[i] = i;
```

```
        }
        std::vector<std::vector<size_t>> orders;
        for (size_t i = 0; i < orders_count; ++i) {
            std::random_shuffle(order.begin(), order.end());
            orders.push_back(order);
        }
        size_t thread_count = 2;
        auto routine = [&graph, &orders](size_t first, size_t last) {
            size_t min = graph.size();
            for (size_t i = first; i < last; ++i) {
                min = std::min(min, _colorize(graph, orders[i]));
            }
            return min;
        };
        // run graph coloring
        std::vector<std::future<size_t>> futures(thread_count);
        for (size_t i = 0; i < thread_count; ++i) {
            size_t len = orders_count / thread_count;
            futures[i] = std::async(std::launch::async, routine, i * len, (i + 1) *
len);
        }
        size_t min = graph.size();
        for (auto& f : futures) {
            min = std::min(min, f.get());
        }

        return min;
    }

    bool _check_coloring(const graph_t& graph, const std::vector<size_t>& coloring) {
        for (int v = 0; v < graph.size(); ++v) {
            for (size_t to : graph[v]) {
                if (coloring[v] == coloring[to]) {
                    return false;
                }
            }
        }
        return true;
    }

    size_t _mex(const std::set<size_t>& set) {
        if (set.empty()) {
            return 0;
        }
        auto max = *set.rbegin();
        for (size_t c = 0; c <= max + 1; ++c) {
            if (set.find(c) == set.end()) {
                return c;
            }
        }
    }

    size_t _colorize(const graph_t& graph, const std::vector<size_t>& order) {
        size_t size = graph.size();
```

```
        std::vector<size_t> colored(size, 0);
        std::vector<size_t> colors(size, 0);
        for (size_t i = 0; i < size; ++i) {
            size_t v = order[i];
            if (!colored[v]) {
                std::set<size_t> used_colors;
                for (auto to : graph[v]) {
                    if (colored[to]) {
                        used_colors.emplace(colors[to]);
                    }
                }
                auto color = _mex(used_colors);
                colored[v] = 1;
                colors[v] = color;
            }
        }
        if (_check_coloring(graph, colors)) {
            std::set<size_t> unique_colors(colors.cbegin(), colors.cend());
            return unique_colors.size();
        } else {
            return graph.size();
        }
    }
}
```

main_improved.cpp

```
#include <iostream>
#include <fstream>
#include <chrono>
#include <iomanip>

#include "algo.h"

using namespace std::chrono;

int main() {
    std::ifstream input("../input.txt");
    std::ofstream output("../output.txt");

    graph_t graph; size_t cnt;
    input >> cnt;
    output << cnt << std::endl;
    for (size_t i = 0; i < cnt; ++i) {
        input >> graph;
        auto start = std::chrono::system_clock::now();
        auto res = improved::colorize(graph);
        auto stop = std::chrono::system_clock::now();
        auto time = duration_cast<milliseconds>(stop - start).count();
        output << std::setw(3) << graph.size() << ' '
               << std::setw(4) << res << ' '
               << time << std::endl;
    }
    return 0;
}
```