Министерство образования и науки Российской Федерации Федеральное агентство по образованию

Федеральное государственное бюджетное образовательное учреждение высшего образования

«Вятский государственный университет»

Факультет автоматики и вычислительной техники

Кафедра электронных вычислительных машин

**Исследование и последовательная реализация вычислительно сложного алгоритма**

Лабораторная работа №1

по курсу «Параллельное программирование»

Вариант 9

Выполнил студент группы ИВТб-31 /Категов А. Д./ Проверил преподаватель /Долженкова М. Л./

Киров 2024

Задание

1. Изучить жадный алгоритм раскраски графа, провести доказательную оценку алгоритма по временной сложности и затратам памяти
2. Проанализировать предыдущий алгоритм и оптимизировать его работу
3. Реализовать обе версии алгоритма с помощью языка C++
4. Построить набор тестовых примеров и провести оценку эффективности реализованных алгоритмов
5. Изучение алгоритма

Жадный алгоритм упорядочивает вершины 𝑣1… 𝑣n в соответствии с некоторой перестановкой и последовательно присваивает вершине 𝑣i наименьший доступный цвет, не использовавшийся для окраски соседей 𝑣i среди 𝑣1 … 𝑣n-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(kn2). Сложность по памяти O(n). Для каждой перестановки вершин (всего 𝑘 перестановок), сгенерированной случайно, внешний цикл перебирает все вершины 𝑣i и для каждой определяет цвета ее соседей, что в худшем случае будет выполнено за 𝑛 шагов.

1. Программная реализация исходного алгоритма

Листинг программной реализации алгоритма приведен в приложении А.

1. Оптимизация исходного алгоритма

Для оптимизации исходного алгоритма были использованы более оптимизированные функции, такие как std::min и vector.assign, и эффективные структуры данных, такие как vector.

Была проведена оптимизация проверок условий с использованием битовых масок. Вместо вектора available и result, разумнее использовать маску, в которой каждый элемент указывает на доступность цвета или покраску вершины. Это позволило значительно ускорить операции проверки условий.

Также быстродействие алгоритма повышает использование типа данных size\_t вместо bool. В vector значения bool упакованы в биты, их сложно кэшировать, а для доступа к элементу может потребоваться несколько команд процессора, что значительно понижает быстродействие.

Данная оптимизация позволила избавится от 2 вложенных циклов, что при больших объемах данных на порядок увеличило быстродействие алгорима. Асимптотическая сложность осталась O(kn2), сложность по памяти O(n).

1. Программная реализация оптимизированного алгоритма

Листинг программной реализации алгоритма приведен в приложении А.

1. Тестирование

Тестирование проводилось на ЭВМ под управлением 64-разрядной OC Windows 10, с 8 ГБ оперативной памяти, с процессором AMD Ryzen 3 4300U 2,70GHz (4 физических ядра, 4 потока). 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) | Время исходного алгоритма, мс | Время оптимизированного алгоритма, мс |
| 5, 7 | 13 | 8 |
| 75, 1000 | 776 | 86 |
| 75, 2500 | 2106 | 156 |
| 90, 300 | 279 | 83 |
| 95, 2500 | 1922 | 151 |
| 150, 5000 | 3779 | 270 |
| 150, 10000 | 7750 | 420 |
| 300, 5000 | 3773 | 375 |
| 300, 10000 | 7485 | 510 |
| 170, 10000 | 7696 | 427 |
| 250, 14000 | 10272 | 584 |
| 250, 17000 | 12693 | 660 |
| 300, 34000 | 25298 | 1320 |
| 400, 40000 | 29120 | 1452 |

**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. Была проделана оптимизация алгоритма для улучшения его быстродействия. В ходе тестирования было замечено, что время работы алгоритма увеличивается как при увеличении числа вершин, так и при увеличении числа ребер. Второй критерий в большей степени влияет на время выполнения алгоритма.

Приложение А

(обязательное)

Листинг программы

#include <vector>

#include <iostream>

#include <cstdint>

#include <set>

#include <unordered\_set>

#include <algorithm>

#include <fstream>

#include <chrono>

#include <iomanip>

using namespace std;

using namespace chrono;

using graph\_t = vector<vector<size\_t>>;

istream& operator>>(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;

}

size\_t \_mex(const vector<size\_t>& set) {

return static\_cast<size\_t>(find(set.begin(), set.end(), 0) - set.begin());

}

size\_t colorize(const graph\_t& graph) {

size\_t size = graph.size();

vector<size\_t> order(size);

size\_t orders\_count = 500;

vector<vector<size\_t>> orders(orders\_count);

for (size\_t i = 0; i < size; ++i) { //O(size)

order[i] = i;

}

for (auto& ord : orders) { //O(orders\_count)

random\_shuffle(order.begin(), order.end());

ord = order;

}

size\_t min = graph.size();

for (const auto& ord : orders) { //O(orders\_count)

vector<size\_t> colored(size, 0); //закрашеные вершины

vector<size\_t> colors(size, 0); //цвета вершин

vector<size\_t> used\_colors(size, 0); //использованые цвета

for (size\_t v : ord) { //O(size)

for (auto to : graph[v]) { //O(size)

if (colored[to]) {

used\_colors[colors[to]] = 1; //этот цвет занят

}

}

auto c = \_mex(used\_colors); //ищем незанятый цвет

colored[v] = 1; //помечаем вершину как раскрашеную

colors[v] = c; //раскрашиваем вершину

used\_colors.assign(size, 0); //сброс всех цветов

}

size\_t colors\_count = 1 + \*max\_element(colors.begin(), colors.end());

min = std::min(min, colors\_count);

}

return min; //O(size + orders\_count + orders\_count \* (size \* size)) = O(orders\_count \* size^2)

}

size\_t colorize2(const graph\_t& graph) {

size\_t size = graph.size();

vector<size\_t> order(size);

size\_t orders\_count = 500;

vector<vector<size\_t>> orders(orders\_count);

for (size\_t i = 0; i < size; ++i) { //O(size)

order[i] = i;

}

for (auto& ord : orders) { //O(orders\_count)

random\_shuffle(order.begin(), order.end());

ord = order;

}

size\_t min = graph.size();

for (const auto& ord : orders) { //O(orders\_count)

vector<int> result(size, -1); // закрашеные вершины

vector<bool> available(size, false); // доступные дли исп цвета

result[0] = 0; // Assign the first color to the first vertex

for (size\_t u : ord) { // Assign colors to the remaining vertices //O(size)

for (int adj : graph[u]) { // Process all adjacent vertices and mark their colors as unavailable //O(size)

if (result[adj] != -1) {

available[result[adj]] = true;

}

}

int color;

for (color = 0; color < size; ++color) { // Find the first available color //O(size)

if (!available[color]) {

break;

}

}

result[u] = color; // Assign the found color to the current vertex

for (int adj : graph[u]) { // Reset the values back to false for the next iteration //O(size)

if (result[adj] != -1) {

available[result[adj]] = false;

}

}

}

size\_t colors\_count = 1 + \*max\_element(result.begin(), result.end());

min = std::min(min, colors\_count);

}

return min; //O(size + orders\_count + orders\_count \* (size \* (size + size + size))) = O(orders\_count \* 3size^2)

}

int main() {

ifstream input("input.txt");

ofstream output("output.txt");

graph\_t graph;

size\_t cnt;

input >> cnt;

output << cnt << endl;

for (size\_t i = 0; i < cnt; ++i) {

input >> graph;

auto start = chrono::system\_clock::now();

auto res = colorize(graph);

auto stop = chrono::system\_clock::now();

auto time = duration\_cast<milliseconds>(stop - start).count();

output << res << endl;

cout << time << endl;

}

input.seekg(0, ios::beg);

input >> cnt;

for (size\_t i = 0; i < cnt; ++i) {

input >> graph;

auto start = chrono::system\_clock::now();

auto res = colorize2(graph);

auto stop = chrono::system\_clock::now();

auto time = duration\_cast<milliseconds>(stop - start).count();

output << res << endl;

cout << time << endl;

}

return 0;

}