Московский Авиационный Институт

(Национальный Исследовательский Университет)

Институт №8 “Компьютерные науки и прикладная математика”

Кафедра №806 “Вычислительная математика и программирование”

**Курсовая работа по курсу**

**«Операционные системы»**

Группа: М8О-215Б-23

Студент: Тараскаев Д.М.

Преподаватель: Миронов Е.С.

Оценка: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Дата: 12.12.24

Москва, 2024

**Постановка задачи**

На языке C\C++ написать программу, которая по конфигурационному файлу в формате json принимает спроектированный DAG джобов и проверяет на корректность:

* отсутствие циклов
* наличие только одной компоненты связности
* наличие стартовых и завершающих джоб

Структура описания джоб и их связей произвольная.

**Общий метод и алгоритм решения**

Создается DAG, который позволяет выполнять задачи с учетом их зависимостей. Отсутствие циклов проверяется с помощью поиска в глубину. Проверка на существование единственной компоненты связности происходит с помощью поиска в ширину.

**Код программы**

**main.cpp**

#include <iostream>

#include <fstream>

#include <vector>

#include <string>

#include <unordered\_map>

#include <unordered\_set>

#include <queue>

#include <nlohmann/json.hpp>

using json = nlohmann::json;

struct Job {

std::string id;

std::vector<std::string> dependencies; // ID джобов, от которых зависит текущий

std::vector<std::string> next\_jobs; // ID джобов, которые зависят от текущего

bool completed = false;

bool failed = false;

bool execute() {

std::cout << "Выполняется джоб " << id << std::endl;

if (failed) {

return false;

}

return true;

}

};

class DAGScheduler {

public:

// Загрузка DAG из JSON-файла

bool loadFromJson(const std::string& filename) {

try {

std::ifstream file(filename);

if (!file.is\_open()) {

std::cerr << "Ошибка открытия файла: " << filename << std::endl;

return false;

}

json config;

file >> config;

// Создаем джобы

for (const auto& job\_json : config["jobs"]) {

std::string id = job\_json["id"];

Job\* job = new Job();

job->id = id;

job->failed = job\_json["failed"];

if (job\_json.contains("dependencies")) {

for (const auto& dep : job\_json["dependencies"]) {

job->dependencies.push\_back(dep);

}

}

jobs[id] = job;

}

// Устанавливаем связи между джобами

for (auto& pair : jobs) {

auto job = pair.second;

for (const auto& dep\_id : job->dependencies) {

if (jobs.find(dep\_id) == jobs.end()) {

std::cerr << "Ошибка: зависимость " << dep\_id << " для джоба " << job->id << " не существует" << std::endl;

return false;

}

jobs[dep\_id]->next\_jobs.push\_back(job->id);

}

}

// Определение стартовых и конечных джобов

for (const auto& pair : jobs) {

if (pair.second->dependencies.empty()) {

start\_jobs.push\_back(pair.first);

}

if (pair.second->next\_jobs.empty()) {

end\_jobs.push\_back(pair.first);

}

}

return true;

}

catch (const std::exception& e) {

std::cerr << "Ошибка при разборе JSON: " << e.what() << std::endl;

return false;

}

}

// Проверка на отсутствие циклов с помощью поиска в глубину

bool checkNoCycles() {

std::unordered\_map<std::string, int> visited;

for (const auto& pair : jobs) {

visited[pair.first] = 0;

}

for (const auto& pair : jobs) {

if (visited[pair.first] == 0) {

if (hasCycleDFS(pair.first, visited)) {

return false;

}

}

}

return true;

}

// Проверка на наличие только одной компоненты связанности

bool checkSingleComponent() {

if (jobs.empty()) return true;

// Создаем неориентированный граф (игнорируем направление связей)

std::unordered\_map<std::string, std::vector<std::string>> undirected\_graph;

for (const auto& pair : jobs) {

undirected\_graph[pair.first] = {};

for (const auto& dep : pair.second->dependencies) {

undirected\_graph[pair.first].push\_back(dep);

undirected\_graph[dep].push\_back(pair.first);

}

}

// BFS для проверки связности

std::unordered\_set<std::string> visited;

std::queue<std::string> q;

// Начинаем с первого джоба

auto first\_job = jobs.begin()->first;

q.push(first\_job);

visited.insert(first\_job);

while (!q.empty()) {

auto current = q.front();

q.pop();

for (const auto& neighbor : undirected\_graph[current]) {

if (visited.find(neighbor) == visited.end()) {

visited.insert(neighbor);

q.push(neighbor);

}

}

}

return visited.size() == jobs.size();

}

bool checkStartAndEndJobs() {

return !start\_jobs.empty() && !end\_jobs.empty();

}

bool validateDAG() {

if (!checkNoCycles()) {

std::cerr << "Ошибка: DAG содержит циклы" << std::endl;

return false;

}

if (!checkSingleComponent()) {

std::cerr << "Ошибка: DAG содержит несколько компонент связанности" << std::endl;

return false;

}

if (!checkStartAndEndJobs()) {

std::cerr << "Ошибка: DAG не содержит стартовых или конечных джобов" << std::endl;

return false;

}

std::cout << "DAG корректен" << std::endl;

return true;

}

bool executeDAG() {

std::unordered\_map<std::string, int> in\_degree;

std::queue<std::string> q;

// Инициализация входящих степеней

for (const auto& pair : jobs) {

in\_degree[pair.first] = pair.second->dependencies.size();

if (in\_degree[pair.first] == 0) {

q.push(pair.first);

}

}

while (!q.empty()) {

std::string current\_id = q.front();

q.pop();

Job\* current\_job = jobs[current\_id];

bool success = current\_job->execute();

current\_job->completed = true;

if (!success) {

std::cerr << "Джоб " << current\_id << " завершился с ошибкой, прерываем выполнение DAG" << std::endl;

return false;

}

// Обработка следующих джобов

for (const auto& next\_id : current\_job->next\_jobs) {

in\_degree[next\_id]--;

if (in\_degree[next\_id] == 0) {

q.push(next\_id);

}

}

}

// Проверяем, что все джобы выполнены

for (const auto& pair : jobs) {

if (!pair.second->completed) {

std::cerr << "Не все джобы были выполнены, возможно есть цикл" << std::endl;

return false;

}

}

std::cout << "DAG выполнен успешно" << std::endl;

return true;

}

~DAGScheduler() {

// Очистка памяти

for (auto& pair : jobs) {

delete pair.second;

}

}

private:

std::unordered\_map<std::string, Job\*> jobs;

std::vector<std::string> start\_jobs; // ID стартовых джобов

std::vector<std::string> end\_jobs; // ID конечных джобов

// Вспомогательная функция для поиска циклов

bool hasCycleDFS(const std::string& id, std::unordered\_map<std::string, int>& visited) {

visited[id] = 1; // В процессе обхода

for (const auto& next\_id : jobs[id]->next\_jobs) {

if (visited[next\_id] == 1) {

return true; // Найден цикл

}

if (visited[next\_id] == 0 && hasCycleDFS(next\_id, visited)) {

return true;

}

}

visited[id] = 2; // Обработан

return false;

}

};

int main(int argc, char\* argv[]) {

if (argc < 2) {

std::cerr << "Использование: " << argv[0] << " <путь\_к\_json\_файлу>" << std::endl;

return 1;

}

std::string json\_file = argv[1];

DAGScheduler scheduler;

if (!scheduler.loadFromJson(json\_file)) {

std::cerr << "Ошибка загрузки конфигурации DAG" << std::endl;

return 1;

}

if (!scheduler.validateDAG()) {

std::cerr << "DAG некорректен" << std::endl;

return 1;

}

if (!scheduler.executeDAG()) {

std::cerr << "Выполнение DAG прервано из-за ошибки" << std::endl;

return 1;

}

return 0;

}

**Протокол работы программы**

**Тестирование**

$ ./main ../src/test1.json

DAG is valid

Executing job job1

Executing job job3

Job job3 failed, aborting DAG execution

DAG execution aborted due to an error

**Вывод**

Программа успешно демонстрирует работу планировщика с последовательным выполнением задач.