

Отчет к дз 3 по C++. Поляков Данила, группа 232

Оптимизировать будем работу шаблонного класса Simulation для типов Velocity, VelocityFlow и P Fixed<64,10>. Для того, чтобы сверить результат работы оптимизации, будем использовать такой тест, прикрепленный в репозитории.

Для оценки времени работы каждой части программы добавим к классу структуру TimeMeasurer, а также разместим в телах функций вызовы std::clock для определения текущего времени.

```
struct TimeMeasurer {
    bool active = false;
    int max_tick;

    clock_t start_time = 0;
    clock_t finish_time = 0;

    clock_t propagate_flow_number = 0;
    clock_t sum_pf_time = 0;

    clock_t propagate_stop_number = 0;
    clock_t sum_ps_time = 0;

    clock_t move_prob_number = 0;
    clock_t sum_mp_time = 0;

    clock_t propagate_move_number = 0;
    clock_t sum_pm_time = 0;

    clock_t sum_cycle1 = 0;
    clock_t sum_cycle2 = 0;
    clock_t sum_cycle3 = 0;
    clock_t sum_cycle4 = 0;

    TimeMeasurer () {
        max_tick = 1000;
    }

    void start() {
        start_time = clock();
    }
}
```

```

void finish() {
    finish_time = clock();
}

void print_results() {
    cout << "Simulation time: " << ((double)finish_time -
start_time) / CLOCKS_PER_SEC << " seconds\n";
    cout << "Average tick time: " << ((double)finish_time -
start_time) / CLOCKS_PER_SEC / max_tick << " seconds\n\n";

    cout << "Cycle1 time per tick: " << (double)sum_cycle1 /
CLOCKS_PER_SEC / max_tick << "; Total cycle1 time: " <<
(double)sum_cycle1 / CLOCKS_PER_SEC << "\n\n";
    cout << "Cycle2 time per tick: " << (double)sum_cycle2 /
CLOCKS_PER_SEC / max_tick << "; Total cycle2 time: " <<
(double)sum_cycle2 / CLOCKS_PER_SEC << "\n";
    cout << "In cycle2 we call propagate_flow" << '\n';
    cout << "Propagate flow calls: " << propagate_flow_number
<< "; Average propagate flow time per tick: "<< (double)sum_pf_time /
CLOCKS_PER_SEC / max_tick << "; Total propagate flow time: " <<
(double)sum_pf_time / CLOCKS_PER_SEC << "\n\n";
    cout << "Cycle3 time per tick: " << (double)sum_cycle3 /
CLOCKS_PER_SEC / max_tick << "; Total cycle3 time: " <<
(double)sum_cycle3 / CLOCKS_PER_SEC << "\n\n";

    cout << "Cycle4 time per tick: " << (double)sum_cycle4 /
CLOCKS_PER_SEC / max_tick << "; Total cycle4 time: " <<
(double)sum_cycle4 / CLOCKS_PER_SEC << "\n";
    cout << "In cycle4 we call propagate_stop, move_prob,
propagate_move" << '\n';
    cout << "Propagate stop calls: " << propagate_stop_number
<< "; Average propagate stop time per tick: "<< (double)sum_ps_time /
CLOCKS_PER_SEC / max_tick << "; Total propagate stop time: " <<
(double)sum_ps_time / CLOCKS_PER_SEC << "\n";
    cout << "Move probability calls: " << move_prob_number <<
"; Average move probability time per tick: "<< (double)sum_mp_time /
CLOCKS_PER_SEC / max_tick << "; Total move probability time: " <<
(double)sum_mp_time / CLOCKS_PER_SEC << "\n";
    cout << "Propagate move calls: " << propagate_move_number
<< "; Average propagate move time per tick: "<< (double)sum_pm_time /
CLOCKS_PER_SEC / max_tick << "; Total propagate move time: " <<
(double)sum_pm_time / CLOCKS_PER_SEC << "\n\n";
}

```

```
};
```

Заметим, что само по себе определение времени несколько замедляет программу, но сохраняется отношение между временами работы отдельных функций класса.

Запустив тест на изначальной не оптимизированной версии программы (по сути копии кода из условия, но вынесенной в класс и с другим типом - `Fixed<64, 10>` вместо `Fixed<32, 16>`), получим следующий результат:

```
Simulation time: 105.674 seconds
Average tick time: 0.105674 seconds

Cycle1 time per tick: 0.00621458; Total cycle1 time: 6.21458

Cycle2 time per tick: 0.0869614; Total cycle2 time: 86.9614
In cycle2 we call propagate_flow
Propagate flow calls: 36842876; Average propagate flow time per tick: 1.84975; Total propagate flow time: 1849.75

Cycle3 time per tick: 0.00607187; Total cycle3 time: 6.07187

Cycle4 time per tick: 0.00552919; Total cycle4 time: 5.52919
In cycle4 we call propagate_stop, move_prob, propagate_move
Propagate stop calls: 2676867; Average propagate stop time per tick: 6.34143; Total propagate stop time: 6341.43
Move probability calls: 3908; Average move probability time per tick: 5.643e-06; Total move probability time: 0.005643
Propagate move calls: 6940; Average propagate move time per tick: 6.1716e-05; Total propagate move time: 0.061716
```

Видим, что на каждом тике большую часть времени выполняется второй цикл, в котором вызывается `propagate_flow`. При этом программа проводит там настолько много времени, что выгоднее всего оптимизировать именно эту часть кода, потому что даже если свести к нулю время работы в остальных функциях, то общее время улучшится лишь на 17 секунд, что кратно меньше, чем время работы `propagate_flow`.

Cycle 2 выглядит следующим образом:

```
do {
    UT += 2;
    prop = 0;

    for (size_t x = 0; x < FieldN; ++x) {
        for (size_t y = 0; y < FieldM; ++y) {
            if (field[x][y] != '#' && last_use[x][y] < UT) {
                auto [t, local_prop, _] = propagate_flow(x, y, 1);
                if (t > 0) {
                    prop = 1;
                }
            }
        }
    }
}
```

```

    }
}

} while (prop);

```

`propagate_flow` - это рекурсивный dfs, который ищет циркуляции в нашей таблице. Сразу видно, в чем проблема: мы m раз вызываем рекурсивную функцию, глубина которой в худшем случае равна $O(n + m)$ (если имеется циркуляция, проходящая вокруг всего поля вдоль стенок). К тому же, наш dfs рекурсивный, поэтому заполняет стек вызовов, что тоже сказывается на времени работы. В идеале стоило бы избавиться от рекурсии вовсе и заменить ее линейным алгоритмом со стеком, однако в данном случае есть еще более существенная проблема. Это обход на каждом тике всего поля и вызовом функции из каждой клетки.

Добавив отладочный вывод следующим образом и запустив программу, можно заметить, что результативные (то есть возвращающий не 0, а некоторое ненулевое значение, свидетельствующее о найденной циркуляции) вызовы функции `propagate_flow` практически единичны.

```

if (t > 0) {
    prop = 1;
    std::cout << x << " " << y << std::endl;
}

```

Кроме того, на каждом следующем шаге циркуляция находится либо в той же клетке, что и на предыдущем шаге, либо в одной из четырех соседних клеток.

На основе этого факта построим такой алгоритм: будем хранить ячейки-кандидаты на запуск dfs-а в `std::vector`. Изначально в него положим все ячейки. Далее на каждом шаге если dfs вернет для некоторой ячейки положительное значение, то положим в `std::vector` для следующего шага ее и 4 ее соседей. В конце каждого шага сделаем swap старого и нового `std::vector`. При этом, в силу зависимости работы всего класса от генератора случайных чисел, может произойти так, что число точек-кандидатов уменьшится до 0. В таком случае можно снова заполнить `std::vector` всеми точками поля. Итоговый вид алгоритма таков:

```

do {
    UT += 2;
    prop = 0;

```

```

        int ln = flow_candidates.size();

        set<pair<int, int>> inside_queue;
        vector <pair<int, int>> newq;

        for (auto [x, y] : flow_candidates){
            if (last_use[x][y] != UT) {
                auto [t, local_prop, _] = propagate_flow(x, y,
1);

                if (t > 0) {
                    prop = 1;

                    if (inside_queue.count({x, y}) == 0) {
                        newq.push_back({x, y});
                        inside_queue.insert({x, y});
                    }

                    vector<pair<int, int>> new_delt{{{ -1, 0},
{1, 0}, {0, -1}, {0, 1}}};

                    for (int u = 1; u < 3; u++) {
                        for (auto [dx, dy]: new_delt) {
                            int nx = x + dx * u, ny = y + dy *
u;

                            if (nx > 0 && ny > 0 && nx < FieldN
&& ny < FieldM && field[nx][ny] != '#') {
                                if (inside_queue.count({nx,
ny}) == 0) {

                                    newq.push_back({nx, ny});
                                    inside_queue.insert({nx,
ny});

                                }

                            }

                        }

                    }

                } else if (last_use[x][y] == UT) {
                    if (inside_queue.count({x, y}) == 0) {
                        newq.push_back({x, y});
                        inside_queue.insert({x, y});
                    }
                }
            }
        }
    }
}

```

```

    }

    }

    }

    swap(newq, flow_candidates);
    inside_queue.clear();
    newq.clear();
    if (flow_candidates.size() < 75) {
        for (size_t x = 0; x < FieldN; ++x) {
            for (size_t y = 0; y < FieldM; ++y) {
                flow_candidates.push_back({x, y});
            }
        }
    }
} while (prop);

```

Запустив этот алгоритм, получим такое время работы:

```

Simulation time: 75.2699 seconds
Average tick time: 0.0752699 seconds

Cycle1 time per tick: 0.00200779; Total cycle1 time: 2.00779

Cycle2 time per tick: 0.0684933; Total cycle2 time: 68.4933
In cycle2 we call propagate_flow
Propagate flow calls: 70634333; Average propagate flow time per tick: 2.00354; Total propagate flow time: 2003.54

Cycle3 time per tick: 0.00231221; Total cycle3 time: 2.31221

Cycle4 time per tick: 0.00217788; Total cycle4 time: 2.17788
In cycle4 we call propagate_stop, move_prob, propagate_move
Propagate stop calls: 2677248; Average propagate stop time per tick: 2.43635; Total propagate stop time: 2436.35
Move probability calls: 4455; Average move probability time per tick: 2.413e-06; Total move probability time: 0.002413
Propagate move calls: 9252; Average propagate move time per tick: 3.7801e-05; Total propagate move time: 0.037801

```

Получили ускорение на 30 секунд.

Обратим внимание на другую часть кода:

```

T &get(int x, int y, int dx, int dy) {
    size_t i = ranges::find(deltas, pair(dx, dy)) -
deltas.begin();
    assert(i < deltas.size());
    return v[x][y][i];
}

```

Структура данных `velocity` представляет собой четыре двумерных массива, которые соответствуют проекциям скорости на каждую из 4 полуосей координат. для поиска нужной из этих 4 полуосей нужной используется `std::ranges`. В целом, библиотека `ranges` призвана упростить и ускорить работу с

подобными контейнерами, однако в данном случае она является не лучшим выбором. Я не погружался в код этой библиотеки, но можно предположить, что `ranges::find` использует под капотом некоторую структуру данных, позволяющую выполнить поиск нужного элемента быстрее, чем линейно (например, какое-то дерево). Из курса алгоритмов и структур данных известно, что такие структуры имеют довольно большую константу, поэтому они начинают работать эффективнее тривиальных алгоритмов лишь начиная с некоторого n . Кроме того, для совсем маленьких значений эти структуры данных могут работать даже хуже, чем наивные алгоритмы, причем кратно хуже. В нашем случае, мы ищем нужное нам значение из 4-х возможных. Попробуем это сделать быстрее, чем линейно, используя бинарные операции над числами (как известно, они работают очень быстро):

```
T &get(int x, int y, int dx, int dy) {  
    int u = ((dy & 1) << 1) | (((dx & 1) & ((dx & 2) >> 1)) |  
    ((dy & 1) & ((dy & 2) >> 1)));  
    assert(u < deltas.size());  
    return v[x][y][u];  
}
```

По сути мы “собрали” индекс в массиве `deltas` из битов `dx`, `dy`. Оставим `assert` для сохранения свойств исходной функции и для безопасности.

Так как мы обращаемся к массиву `velocity` практически в каждой функции, это улучшение уменьшит время работы в 2 раза.

Теперь прокомментируем все измерения времени, кроме измерения общего времени работы и проведем финальные замеры времени выполнения 1000 тиков.

1) Версия программы без модификаций:

[illegible]

2) Версия программы с двумя оптимизациями:

[illegible]

3) Версия программы только с оптимизацией velocity.get:

[illegible]

Видим, что первая оптимизация слегка меняет поведение программы (тяжелая жидкость в центре не успела за 1000 тиков утонуть в легкой жидкости). При этом первая оптимизация замедляет программу на данном тесте. Это связано с тем, что тест сравнительно небольшой. Поэтому в финальной версии программы эта часть закомментирована.

Таким образом, мы уменьшили время работы программы в два раза, не используя при этом потоки. В файле Simulator.h приведена версия класса с функциями измерения времени в каждой функции, а также итоговая, самая быстрая версия класса. В ней закомментирована первая оптимизация. Её можно включать на больших тестах вместо простого обхода двумерного массива при необходимости.