

МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ
по лабораторной работе №1
по дисциплине «Построение и анализ алгоритмов»
Тема: Поиск с возвратом

Студент гр. 8304

Чешуин Д.И.

Преподаватель

Размочаева Н.В.

Санкт-Петербург

2020

Цель работы.

Ознакомиться с алгоритмом поиска с возвратом, научиться оценивать временную сложность алгоритма и применять его для решения задач.

Постановка задачи.

Вариант 1и. Итеративный бэктрекинг. Поиск решения за разумное время (меньше минуты) для $2 \leq N \leq 30$.

Входные данные:

Размер столешницы – одно целое число $N(2 \leq N \leq 20)$.

Выходные данные:

Одно число задающее минимально количество обрезков (квадратов), из которых можно построить столешницу (квадрат) заданного размера N . Далее должны идти K строк, каждая из которых должна содержать три целых числа x , y и w , задающие координаты левого верхнего угла ($1 \leq x, y \leq N$) и длину стороны соответствующего обрезка (квадрата).

Описание алгоритма.

Алгоритм разбиения:

Т.к. наименьшее разбиение для чисел, не являющихся простыми, будет совпадать с их наименьшим простым делителем, сначала мы находим этот делитель. Затем с помощью некоторой функции разбиения получаем начальную конфигурацию, которую затем пытаемся улучшить с помощью бэктрекинга.

Алгоритм бэктрекинга пытается заполнить изначальный квадрат квадратами наибольшего размера, затем удаляет все последние единичные квадраты и уменьшает последний не единичный. После этого цикл повторяется. Алгоритм завершает работу, когда не останется квадратов, размер которых можно уменьшить.

Оптимизации алгоритма.

- 1) Было обнаружено, что у всех минимальных разбиений совпадают первые 3 части, за счёт этого значительно снижается число проверяемых вариантов.
- 2) Отбрасываются все варианты, частичное решение которых содержит большее или равное количество частей относительно текущего лучшего варианта.
- 3) Отбрасываются симметричные варианты, за счёт ограничения области добавления новых квадратов и запылнения оставшегося свободного места наибольшими возможными квадратами.

- 4) Придуман алгоритм генерации базовой конфигурации, который даёт первоначальное решение близкое к идеальному, а в ряде случаев - идеальное решение, что отбрасывает большое количество плохих разбиений за счёт оптимизации номер 2.

Анализ алгоритма.

Для квадратов, сторона которых не является простым числом алгоритм работает примерно за одно и то же время, что и для квадрата со стороной равной минимальному простому делителю числа. Сложность алгоритма по времени возрастает по экспоненте. Сложность по памяти $O(N^2)$

Описание функций и СД.

Для решения задачи был реализован класс Table и структура отдельной части - Part

Класс содержит методы вывода на экран промежуточных решений, получения минимального разбиения, генерации начального решения.

Промежуточные решения хранятся в двумерном массиве, а перечень квадратов в векторе.

Метод бэктрекинга:

`bool verificateWithBacktracking ()`

Ничего не принимает т.к. использует поля класса, возвращает true, если было найдено решение, лучше начальной конфигурации, иначе false. Функция записывает промежуточные данные и результат в поля класса.

`bool hasSpaceTo(Part part)`

Метод проверяем, хватит ли места для переданной части.

`bool addNewPart()`

Метод добавляет новую часть и возвращает true. Если стол уже заполнен – возвращает false.

`bool reduceLastImportantPart()`

Метод уменьшает последнюю значимую часть и удаляет все предыдущие. Возвращает false, если не осталось частей для уменьшения, иначе – true.

`bool removeLastPart()`

Удаляет последнюю часть. Возвращает false, если не осталось частей для удаления, иначе – true.

`void printConfiguration()`

Метод печатает на экран текущую конфигурацию стола.

`vector<Part> createStartingConfiguration()`

Метод генерирует начальную конфигурацию и возвращает её.

`vector<Part> getConfiguration()`

Метод возвращает текущую конфигурацию.

`unsigned long long getItersCount()`

Метод возвращает число итераций, затраченное методом бэктрекинга.

Спецификация программы.

Программа предназначена для нахождения минимального способа разбиения квадрата на меньшие квадраты. Программа написана на языке C++. Входными данными является число N (сторона квадрата), выходными – минимальное количество меньших квадратов и K строк, содержащие координаты левого верхнего угла и длину стороны соответствующего квадрата.

Выводы.

В ходе выполнения лабораторной работы был реализован алгоритм итеративного бэктрекинга, дана оценка времени работы алгоритма, а также были получены навыки решения задач с помощью поиска с возвратом.

ПРИЛОЖЕНИЕ А. ИСХОДНЫЙ КОД ПРОГРАММЫ

main.cpp.

```
#include <iostream>
#include <queue>
#include <vector>
#include <chrono>

using namespace std;

class Table
{
public:
    struct Part
    {
        unsigned x = 0;
        unsigned y = 0;
        unsigned size = 0;
    };
private:
    unsigned _size = 0;
    vector<vector<uint8_t>> _cells;
    unsigned _searchPos = 0;
    vector<Part> _bestConfiguration;
    vector<Part> _parts;
    queue<Part> _onCheck;
    queue<Part> _onSplit;
    unsigned long long _itersCount = 0;
private:
    bool hasSpaceTo(Part part);
    void checkPart();
    void splitPart();
    bool addNewPart();
    bool reduceLastImportantPart();
    bool removeLastPart();
    void printConfiguration();
public:
    Table(unsigned size);
    vector<Part> createStartingConfiguration();
    vector<Part> getConfiguration();
    bool verificateWithBacktracking();
    unsigned long long getItersCount();
};

int main()
{
    unsigned size = 0;
    unsigned divider = 0;
    unsigned multiplier = 0;
    cin >> size;
    for(unsigned i = 2; i <= size; i++)
    {
        if(size % i == 0)
        {
            divider = i;
```

```

        multiplier = size / divider;
        break;
    }
}

Table table(divider);
vector<Table::Part> solution;

auto start = std::chrono::system_clock::now();

solution = table.createStartingConfiguration();
if(table.verificateWithBacktracking())
{
    solution = table.getConfiguration();
}

auto end = std::chrono::system_clock::now();

auto delta = std::chrono::duration_cast<std::chrono::microseconds>(end-start).count();

cout << "Search finished!" << endl;
cout << "Parts count - " << solution.size() << endl;
unsigned i = 1;
for(auto part : solution)
{
    cout <<"Part - " << i << " x - " << part.x * multiplier << " y - "
        << part.y * multiplier << " size - " << part.size * multiplier << endl;
    i += 1;
}

cout << "Time - " << static_cast<float>(delta) / 1000000 << endl;
cout << "Iterations - " << table.getItersCount() << endl;
return 0;
}

Table::Table(unsigned size)
{
    _size = size;
    _cells.resize(size);
    for (unsigned i = 0; i < size; i++)
    {
        _cells[i].resize(size);
        for (unsigned j = 0; j < size; j++)
        {
            _cells[i][j] = 0;
        }
    }

    Part starter;
    starter.x = 0;
    starter.y = 0;
    starter.size = size;

    _onSplit.push(starter);
}

vector<Table::Part> Table::createStartingConfiguration()
{
    while (!(_onCheck.empty() && _onSplit.empty()))

```

```

    {
        while(!_onCheck.empty())
        {
            checkPart();
        }
        while(!_onSplit.empty())
        {
            splitPart();
        }
    }

    _bestConfiguration = _parts;

    return _bestConfiguration;
}

bool Table::hasSpaceTo(Part part)
{
    for(unsigned x = part.x; x < part.x + part.size; x++)
    {
        for(unsigned y = part.y; y < part.y + part.size; y++)
        {
            if(_cells[y][x])
            {
                return false;
            }
        }
    }

    return true;
}

void Table::checkPart()
{
    Part part = _onCheck.front();
    _onCheck.pop();
    if(hasSpaceTo(part))
    {
        for(unsigned x = part.x; x < part.x + part.size; x++)
        {
            for(unsigned y = part.y; y < part.y + part.size; y++)
            {
                uint8_t partNum = _parts.size() + 1;
                _cells[y][x] = partNum;
            }
        }
        _parts.push_back(part);
    }
    else
    {
        _onSplit.push(part);
    }
}

void Table::splitPart()
{
    Part part = _onSplit.front();
    _onSplit.pop();
    if(part.size <= 1)

```

```

{
    return;
}

unsigned isOdd = part.size % 2;
for(unsigned onLeft = 0; onLeft < 2; onLeft++)
{
    for(unsigned onTop = 0; onTop < 2; onTop++)
    {
        Part newPart;
        newPart.x = part.x + (part.size / 2 + isOdd) * onLeft - isOdd * (onLeft & onTop);
        newPart.y = part.y + (part.size / 2 + isOdd) * onTop - isOdd * (onLeft & onTop);
        newPart.size = part.size / 2 + isOdd * (1 - onLeft ^ onTop);

        _onCheck.push(newPart);
    }
}

vector<Table::Part> Table::getConfiguration()
{
    return _bestConfiguration;
}

bool Table::addNewPart()
{
    _itersCount += 1;

    unsigned maxPos = _size * _size;
    for(; _searchPos < maxPos; _searchPos++)
    {
        unsigned y0 = _searchPos / _size;
        unsigned x0 = _searchPos % _size;

        if(_cells[y0][x0] == 0)
        {
            //проверяем, часть какого размера войдёт на это место
            unsigned freeSpace = 0;
            bool hasSpace = true;
            while(hasSpace)
            {
                freeSpace += 1;
                if(x0 + freeSpace >= _size || y0 + freeSpace >= _size)
                {
                    hasSpace = false;
                    break;
                }
                for(unsigned x = x0; x < x0 + freeSpace; x++)
                {
                    if(_cells[y0 + freeSpace][x])
                    {
                        hasSpace = false;
                        break;
                    }
                }
                for(unsigned y = y0; y < y0 + freeSpace; y++)
                {
                    if(_cells[y][x0 + freeSpace])
                    {

```



```

        hasSpace = false;
        break;
    }
}

//добавляем новую часть
Part part;
part.size = freeSpace;
part.x = x0;
part.y = y0;

for(unsigned x = part.x; x < part.x + part.size; x++)
{
    for(unsigned y = part.y; y < part.y + part.size; y++)
    {
        uint8_t partNum = _parts.size() + 1;
        _cells[y][x] = partNum;
    }
}

_searchPos += part.size;
_parts.push_back(part);

return true;
}

return false;
}

bool Table::reduceLastImportantPart()
{
    _itersCount += 1;

    unsigned border = _size / 2 + _size / 4 + 1;
    Part part = _parts.back();
    while(part.y > border)
    {
        if(!removeLastPart())
        {
            return false;
        }

        part = _parts.back();
    }
    if(_parts.back().size > 1)
    {
        _parts.pop_back();

        for(unsigned y = part.y; y < part.y + part.size; y++)
        {
            _cells[y][part.x + part.size - 1] = 0;
        }
        for(unsigned x = part.x; x < part.x + part.size; x++)
        {
            _cells[part.y + part.size - 1][x] = 0;
        }
    }
}

```

```

        part.size -= 1;
        _parts.push_back(part);
        _searchPos = part.y * _size + part.x + part.size;

        return true;
    }
    else
    {
        return false;
    }
}

bool Table::removeLastPart()
{
    _itersCount += 1;

    //первые 3 части гарантировано корректны, менять их не имеет смысла
    if(_parts.size() > 3)
    {
        Part part = _parts.back();
        _parts.pop_back();

        for(unsigned x = part.x; x < part.x + part.size; x++)
        {
            for(unsigned y = part.y; y < part.y + part.size; y++)
            {
                _cells[y][x] = 0;
            }
        }

        _searchPos = part.y * _size + part.x;

        return true;
    }
    else
    {
        return false;
    }
}

bool Table::verificateWithBacktracking()
{
    bool hasBetterSolution = false;

    cout << "Starting configuration generated!" << endl;
    printConfiguration();
    cout << "-----Trying to find better solution...-----"
    << endl;

    //очистка текущей конфигурации до 3 частей
    while(removeLastPart());

    _searchPos = 0;

    bool allChecked = false;
    bool isFull = false;
    while (!allChecked)
    {
        while(!isFull && _parts.size() < _bestConfiguration.size())

```

```

    {
        isFull = !addNewPart();
    }

    if(isFull)
    {
        _bestConfiguration = _parts;
        hasBetterSolution = true;

        cout << "Better solution finded!" << endl;
        cout << "Parts count - " << _bestConfiguration.size() << endl;
        printConfiguration();
        cout << "-----Trying to find better solution...-----
-----" << endl;
    }

    while(!reduceLastImportantPart())
    {
        if(!removeLastPart())
        {
            allChecked = true;
            break;
        }
    }
    isFull = false;

}

return hasBetterSolution;
}

void Table::printConfiguration()
{
    for(unsigned y = 0; y < _size; y++)
    {
        for(unsigned x = 0; x < _size; x++)
        {
            cout.width(2);
            cout.fill(' ');
            cout << static_cast<unsigned>(_cells[y][x]) << " ";
        }
        cout << endl << endl;
    }
}

unsigned long long Table::getItersCount()
{
    return _itersCount;
}

```