**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра МОЭВМ**

отчет

**по лабораторной работе №1**

**по дисциплине «Проектирование и анализ алгоритмов»**

Тема: Поиск с возвратом

|  |  |  |
| --- | --- | --- |
| Студент гр. 6304 |  | Запевалов А.И. |
| Преподаватель |  | Филатов А.Ю. |

Санкт-Петербург

2019

**Цель работы.**

**Входные данные**

Размер столешницы - одно целое число *N*(2≤*N*≤20).   
**Выходные данные**

Одно число *K*, задающее минимальное 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≤*x*, *y*≤*N*) и длину стороны соответствующего обрезка(квадрата).

**Основные теоретические положения.**

Поиск с возвратом, бэктрекинг (англ. backtracking) — общий метод нахождения решений задачи, в которой требуется полный перебор всех возможных вариантов в некотором множестве М.

Решение задачи методом поиска с возвратом сводится к последовательному расширению частичного решения. Если на очередном шаге такое расширение провести не удается, то возвращаются к более короткому частичному решению и продолжают поиск дальше. Данный алгоритм позволяет найти все решения поставленной задачи, если они существуют. Для ускорения метода стараются вычисления организовать таким образом, чтобы как можно раньше выявлять заведомо неподходящие варианты. Зачастую это позволяет значительно уменьшить время нахождения решения.

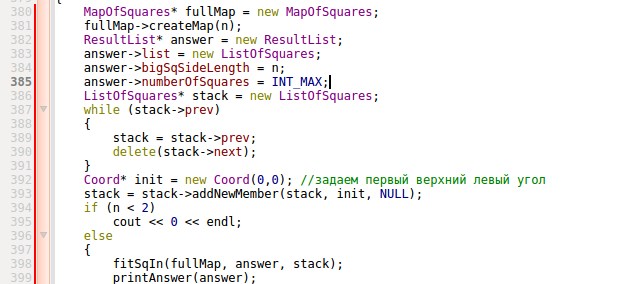
**Экспериментальные результаты.**



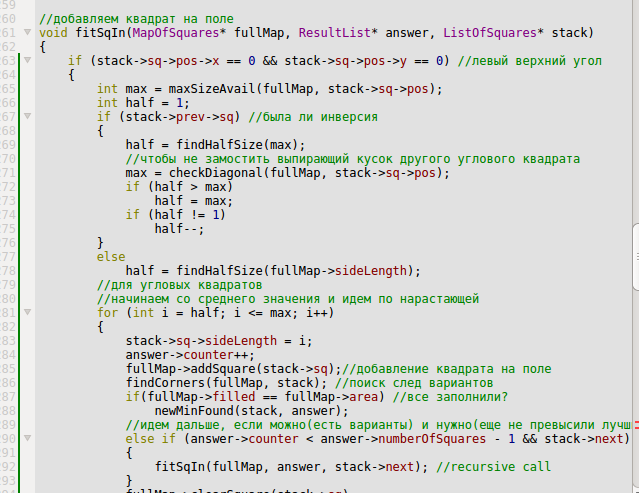
**Обработка результатов эксперимента.**

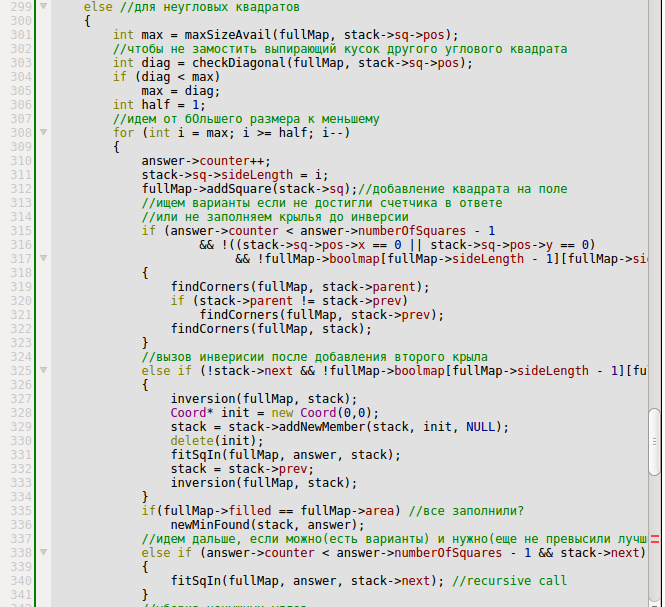
Подход к выполнению задачи.

1. Создаются необходимые структуры:

Класс MapOfSquares содержит двумерный массив переменных типа bool, значение которых соответствует заполненным и свободным ячейкам столешницы. Структура ResultList содержит двунаправленный список указателей на структуры, содержащих указатели на квадраты, а также счетчик текущего квадрата и значение лучшего результата. Двунаправленный список ListOfSquares содержится как и в результирующей структуре, так и используется для заполнения и удаления квадратов в процессе поиска.

2. Основная рекурсивная функция и принцип ее работы.

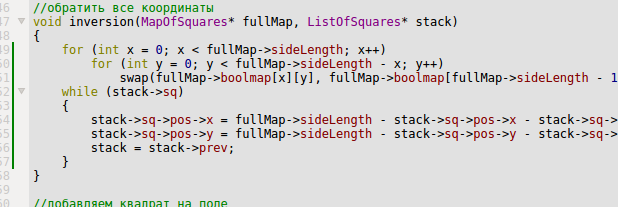
Для сокращения времени выполнения программы функция разделена на две части — для угловых(0,0) и остальных квадратов. Для угловых квадратов варианты обрабатываются от среднего из возможных до максимального или от среднего-1 для квадрата в правом нижнем углу, так как он не всегда оказывается самым большим среди всех квадратов, не имеющих одной из координат 0. Такой ход направления поиска позволяет быстро найти хороший вариант чтобы ограничить глубину поиска и ускорить тем самым работу программы.

Для неугловых квадратов выполняется проверка того, находится ли квадрат с краю, для того чтобы в таком случае было возможно ограничить варианты на текущем шаге одним максимально возможным. Эту проверку проходят все квадраты с краю кроме квадратов смежных со вторым угловым квадратом, т. к. лучший вариант может содержать квадраты с краю, не имеющие максимально допустимый размер. В противоположность случаю с угловым квадратом здесь поиск происходит от максимального размера до 1 для выявления хорошего результата на раннем этапе.

Проверка ограничения глубины происходит когда сравниваются величины answer→counter и answer→numberOfSquares-1.

Также в этом месте происходит вызов инверсии когда текущий квадрат является вторым из смежных с первым угловым квадратом.

3.Инверсия

Одной из эвристик, используемых для ускорения работы программы, является инверсия. Инверсия происходит на этапе когда оказываются заполненными первый угловой квадрат и квадраты, смежные с ним. Это необходимо для повторного выполнения поиска с возвратом начиная с угловой ячейки, т. к. можно повторно ограничить область поиска, заполнив два края максимально возможными квадратами, которые, в большинстве случаев будут содержаться в лучшем варианте. Если поиск оказался неудачным инверсия выполняется снова, чтобы изменить первый угловой квадрат и квадраты, смежные с ним.

**Выводы.**

Был реализован алгоритм поиска с возвратом. Были решены проблемы организации рекурсивного поиска с целью выявления хороших результатов на ранних этапах. Организована работа алгоритма, генерирующего результат с относительно небольшой глубиной для ее ограничения в дальнейшем поиске. Для ускорения работы программы успешно добавлены эвристические функции (инверсия, проход начиная с середины) таким образом, что в их отсутствии программа продолжает выполнять поставленную задачу, но за большее время.

**ПРИЛОЖЕНИЕ А**

**Исходный код программы.**

#include <iostream>

#include "stdlib.h"

#include "limits.h"

#include <fstream>

using namespace std;

class Coord{ //координаты квадрата

public:

int x;

int y;

Coord(int x, int y){

this->x = x, this->y = y;

}

bool operator==(const Coord& a){

return (this->x == a.x && this->y == a.y);

}

};

class Square{ //инфа о квадрате

public:

Coord\* pos = NULL;

int sideLength;

Square(int x, int y, int s){

this->pos = new Coord(x, y);

this->sideLength = s;

}

};

class ListOfSquares //список квадратов

{

public:

Square\* sq = NULL;

ListOfSquares\* parent = NULL;

ListOfSquares\* next = NULL;

ListOfSquares\* prev = NULL;

//добавляется новый квадрат в список, возващается на него указатель

ListOfSquares\* addNewMember(ListOfSquares\* curMember, Coord\* pos, ListOfSquares\* parent)

{

ListOfSquares\* newMember = new ListOfSquares;

while (curMember->next)

curMember = curMember->next;

newMember->parent = parent;

newMember->sq = new Square(pos->x, pos->y, 0);

curMember->next = newMember;

newMember->prev = curMember;

newMember->next = NULL;

return newMember;

}

//удаление квадрата из списка

ListOfSquares\* removeMember(ListOfSquares\* curMember)

{

curMember = curMember->prev;

delete(curMember->next->sq->pos);

delete(curMember->next->sq);

delete(curMember->next);

curMember->next = NULL;

return curMember;

}

};

struct ResultList //ответ

{

int bigSqSideLength;

int numberOfSquares;

int counter;

ListOfSquares\* list = NULL;

};

class MapOfSquares //карта заполнения

{

public:

bool\*\* boolmap = NULL;

int sideLength;

int filled;

int area;

void createMap(int n) //false - не заполнено, true - заполнено

{

sideLength = n;

area = sideLength \* sideLength;

boolmap = new bool\*[n];

filled = 0;

/\*for (bool\* s : boolmap)

{

s = new bool[n];

for (bool t : s)

t = false;

}\*/

for (int i = 0; i < n; i++)

{

boolmap[i] = new bool[n];

for (int j = 0; j < n; j++)

boolmap[i][j] = false;

}

}

void printMap() //вывод

{

for (int i = 0; i < sideLength; i++)

{

for (int j = 0; j < sideLength; j++)

cout << boolmap[i][j];

cout << " " << endl;

}

}

void deleteMap() //удаление

{

for (int i = 0; i < sideLength; i++)

delete(boolmap[i]);

delete(this);

}

void addSquare (Square\* sq) //заполнение нужного участка

{

for (int i = 0; i < sq->sideLength; i++)

for (int j = 0; j<sq->sideLength; j++)

this->boolmap[sq->pos->x + i][sq->pos->y + j] = true;

filled += sq->sideLength\*sq->sideLength;

}

void clearSquare (Square\* sq) //очистка нужного участка

{

for (int i = 0; i < sq->sideLength; i++)

for (int j = 0; j<sq->sideLength; j++)

this->boolmap[sq->pos->x + i][sq->pos->y + j] = false;

filled -= sq->sideLength\*sq->sideLength;

}

};

//проверка угловой точки

int checkDiagonal(MapOfSquares\* fullMap, Coord\* leftUpperCorner)

{

int counter = 1;

while (leftUpperCorner->x + counter <= fullMap->sideLength

&& leftUpperCorner->y + counter <= fullMap->sideLength)

{

if (fullMap->boolmap[leftUpperCorner->x + counter - 1][leftUpperCorner->y + counter - 1])

return counter - 1;

counter++;

}

return counter - 1;

}

//проверка точек справа и снизу от текущей

int maxSizeAvail(MapOfSquares\* fullMap, Coord\* leftUpperCorner)

{

int counter = 1;

while (counter < fullMap->sideLength

&& leftUpperCorner->x + counter <= fullMap->sideLength

&& leftUpperCorner->y + counter <= fullMap->sideLength)

{

if (fullMap->boolmap[leftUpperCorner->x + counter - 1][leftUpperCorner->y]

|| fullMap->boolmap[leftUpperCorner->x][leftUpperCorner->y + counter - 1])

return counter - 1; //возврат последней доступной точки

counter++;

}

return counter - 1; //возврат последней доступной точки

}

//был ли ранее внесен в стек предлагаемый угол, чтобы не дублироваться

bool checkCornerInList(Coord\* pos, ListOfSquares\* stack)

{

do{

if (\*pos == \*stack->sq->pos)

return true;

stack = stack->next;

} while (stack);

return false;

}

//проверка двух точек справа и снизу от текущей на то,

//являются ли они левыми верхними углами

//если да, добавляются в стек

void findCorners(MapOfSquares\* fullMap, ListOfSquares\* stack)

{

Coord\* freeCell = new Coord(0,0);

ListOfSquares\* tmp = stack;

if (stack->sq->pos->y + stack->sq->sideLength < fullMap->sideLength

&& (stack->sq->pos->x - 1 < 0

|| fullMap->boolmap[stack->sq->pos->x - 1][stack->sq->pos->y + stack->sq->sideLength])

&& fullMap->boolmap[stack->sq->pos->x][stack->sq->pos->y + stack->sq->sideLength] == false)

{

freeCell->x = stack->sq->pos->x;

freeCell->y = stack->sq->pos->y + stack->sq->sideLength;

if (!checkCornerInList(freeCell, tmp))

tmp = tmp->addNewMember(tmp, freeCell, stack);

}

if (stack->sq->pos->x + stack->sq->sideLength < fullMap->sideLength

&& (stack->sq->pos->y - 1 < 0

|| fullMap->boolmap[stack->sq->pos->x + stack->sq->sideLength][stack->sq->pos->y - 1])

&& fullMap->boolmap[stack->sq->pos->x + stack->sq->sideLength][stack->sq->pos->y] == false)

{

freeCell->x = stack->sq->pos->x + stack->sq->sideLength;

freeCell->y = stack->sq->pos->y;

if (!checkCornerInList(freeCell, tmp))

tmp = tmp->addNewMember(tmp, freeCell, stack);

}

delete(freeCell);

}

//заполнение ответа новым вариантом

void newMinFound(ListOfSquares\* stack, ResultList\* answer)

{

while (answer->list->prev)

{

answer->list = answer->list->prev;

delete(answer->list->next->sq->pos);

delete(answer->list->next->sq);

delete(answer->list->next);

}

answer->list->next = NULL;

while (stack->sq)

{

answer->list = answer->list->addNewMember(answer->list, stack->sq->pos, NULL);

answer->list->sq->sideLength = stack->sq->sideLength;

stack = stack->prev;

}

answer->numberOfSquares = answer->counter;

}

// деление на два с округлением в большую сторону

int findHalfSize(int sideLength)

{

int half;

if (sideLength%2 == 0)

half = sideLength / 2;

else

half = sideLength / 2 + 1;

return half;

}

// уборка ненужных предложенных углов

ListOfSquares\* clearVariants(ListOfSquares\* stack)

{

while (stack->next)

stack = stack->next;

while (!stack->sq->sideLength)

stack = stack->removeMember(stack);

return stack;

}

//обратить все координаты

void inversion(MapOfSquares\* fullMap, ListOfSquares\* stack)

{

for (int x = 0; x < fullMap->sideLength; x++)

for (int y = 0; y < fullMap->sideLength - x; y++)

swap(fullMap->boolmap[x][y], fullMap->boolmap[fullMap->sideLength - 1 - x][fullMap->sideLength - 1 - y]);

while (stack->sq)

{

stack->sq->pos->x = fullMap->sideLength - stack->sq->pos->x - stack->sq->sideLength;

stack->sq->pos->y = fullMap->sideLength - stack->sq->pos->y - stack->sq->sideLength;

stack = stack->prev;

}

}

//добавляем квадрат на поле

void fitSqIn(MapOfSquares\* fullMap, ResultList\* answer, ListOfSquares\* stack)

{

if (stack->sq->pos->x == 0 && stack->sq->pos->y == 0) //левый верхний угол

{

int max = maxSizeAvail(fullMap, stack->sq->pos);

int half = 1;

if (stack->prev->sq) //была ли инверсия

{

half = findHalfSize(max);

//чтобы не замостить выпирающий кусок другого углового квадрата

max = checkDiagonal(fullMap, stack->sq->pos);

if (half > max)

half = max;

if (half != 1)

half--;

}

else

half = findHalfSize(fullMap->sideLength);

//для угловых квадратов

//начинаем со среднего значения и идем по нарастающей

for (int i = half; i <= max; i++)

{

stack->sq->sideLength = i;

answer->counter++;

fullMap->addSquare(stack->sq);//добавление квадрата на поле

findCorners(fullMap, stack); //поиск след вариантов

if(fullMap->filled == fullMap->area) //все заполнили?

newMinFound(stack, answer);

//идем дальше, если можно(есть варианты) и нужно(еще не превысили лучший результат)

else if (answer->counter < answer->numberOfSquares - 1 && stack->next)

{

fitSqIn(fullMap, answer, stack->next); //recursive call

}

fullMap->clearSquare(stack->sq);

answer->counter--;

}

stack = stack->removeMember(stack);

}

else //для неугловых квадратов

{

int max = maxSizeAvail(fullMap, stack->sq->pos);

//чтобы не замостить выпирающий кусок другого углового квадрата

int diag = checkDiagonal(fullMap, stack->sq->pos);

if (diag < max)

max = diag;

int half = 1;

//идем от бОльшего размера к меньшему

for (int i = max; i >= half; i--)

{

answer->counter++;

stack->sq->sideLength = i;

fullMap->addSquare(stack->sq);//добавление квадрата на поле

//ищем варианты если не достигли счетчика в ответе

//или мы не заполняем крылья до инверсии

if (answer->counter < answer->numberOfSquares - 1

&& !((stack->sq->pos->x == 0 || stack->sq->pos->y == 0)

&& !fullMap->boolmap[fullMap->sideLength - 1][fullMap->sideLength - 1]))

{

findCorners(fullMap, stack->parent);

if (stack->parent != stack->prev)

findCorners(fullMap, stack->prev);

findCorners(fullMap, stack);

}

//вызов инверисии после добавления второго крыла

else if (!stack->next && !fullMap->boolmap[fullMap->sideLength - 1][fullMap->sideLength - 1])

{

inversion(fullMap, stack);

Coord\* init = new Coord(0,0);

stack = stack->addNewMember(stack, init, NULL);

delete(init);

fitSqIn(fullMap, answer, stack);

stack = stack->prev;

inversion(fullMap, stack);

}

if(fullMap->filled == fullMap->area) //все заполнили?

newMinFound(stack, answer);

//идем дальше, если можно(есть варианты) и нужно(еще не превысили лучший результат)

else if (answer->counter < answer->numberOfSquares - 1 && stack->next)

{

fitSqIn(fullMap, answer, stack->next); //recursive call

}

//уборка ненужных углов

else

{

stack = clearVariants(stack);

i = 0;

}

fullMap->clearSquare(stack->sq);

answer->counter--;

// для крыльев первого угла достаточно проверки одного размера,

// для крыльев второго - нет

// для последующих квадратов у края достаточно проверки одного размера.

if ((stack->sq->pos->x == 0 || stack->sq->pos->y == 0)

&& !(stack->sq->pos->x == 0 && stack->sq->pos->y == 0)

&& (answer->counter < 3

|| answer->counter > 6))

i = 0;

}

stack = stack->removeMember(stack);

}

}

void printAnswer(ResultList\* answer) //вывод

{

/\*ofstream fout;

fout.open("output.csv", ios\_base::app);

fout << answer->bigSqSideLength << ';' << answer->numberOfSquares << endl;

fout.close();\*/

cout << answer->numberOfSquares << endl;

while (answer->list->prev)

{

cout << answer->list->sq->pos->x + 1 << ' ' << answer->list->sq->pos->y + 1 << ' ' << answer->list->sq->sideLength << endl;

answer->list = answer->list->prev;

}

}

//создаем структуры, задаем начальные координаты

void initiate(int n)

{

MapOfSquares\* fullMap = new MapOfSquares;

fullMap->createMap(n);

ResultList\* answer = new ResultList;

answer->list = new ListOfSquares;

answer->bigSqSideLength = n;

answer->numberOfSquares = INT\_MAX;

answer->counter = 0;

ListOfSquares\* stack = new ListOfSquares;

while (stack->prev)

{

stack = stack->prev;

delete(stack->next);

}

Coord\* init = new Coord(0,0); //задаем первый верхний левый угол

stack = stack->addNewMember(stack, init, NULL);

if (n < 2)

cout << 0 << endl;

else

{

fitSqIn(fullMap, answer, stack);

printAnswer(answer);

}

stack = stack->prev;

delete(init);

delete(stack);

while (answer->list->prev)

{

answer->list = answer->list->prev;

delete(answer->list->next->sq->pos);

delete(answer->list->next->sq);

delete(answer->list->next);

}

delete(answer->list);

delete(answer);

//fullMap->printMap();

fullMap->deleteMap();

}

int main()

{

int n;

cin >> n;

initiate(n);

/\*for (n = 2; n <= 40; n++)

initiate(n);\*/

return 0;

}