**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра Вычислительной техники**

отчет

**по лабораторной работе №1**

**по дисциплине «Введение в искусственный интеллект»**

**Тема: «СТРАТЕГИИ ПОИСКА»**

|  |  |  |
| --- | --- | --- |
| Студенты гр. 9308 |  | Степовик В.С.  Соболев М.С.  Аюпов Р. Н. |
| Преподаватель |  | Родионов С.В. |

Санкт-Петербург

2022г.

**Цель работы**

Практическое закрепление понимания общих идей поиска в пространстве состояний и стратегий слепого поиска.

**Постановка задачи**

Реализовать программу поиска пути решения головоломки «8-ка» с использованием алгоритма поиска в глубину и его модификации – итеративного алгоритма поиска в глубину – для двух заданных состояний: целевого и исходного. Экспериментальным путем оценить временную и емкостную сложность решения задачи для двух заданных стратегий.

Описанные выше состояния изображены на рисунке 1 и 2 соответственно.

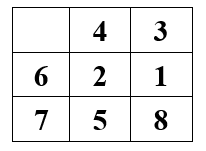


Рис. 1. Начальное состояние

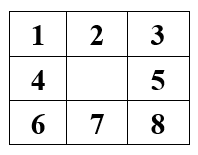


Рис. 2. Конечное состояние

**Распределение обязанностей:**

Аюпов Ренат – реализация графического интерфейса, сервисных функций и взаимодействия со структурами данных.

Соболев Матвей – реализация алгоритма итеративного поиска в глубину и построение архитектуры программы.

Степовик Виктор – реализация алгоритма поиска в глубину, функций пошагового вывода результата, и сравнительная оценка сложности.

**Описание выбранных структур данных**

Для решения поставленной задачи было решено использовать язык программирования C++, поэтому программа получила реализацию в парадигме ООП. Два представленных ниже класса реализуют дерево состояний и его вершины соответственно.

|  |  |  |
| --- | --- | --- |
| Структура | Поля | Описание |
| class Node | char\*\* elements;  int depth;  struct Node\* left;  struct Node\* mid\_left;  struct Node\* mid\_right;  struct Node\* right;  struct Node\* parent; | Является элементом дерева, содержит массив элементов, который является слепком состояния «8-ки», ссылки на потомков и родителя (каждый из потомков существует, если есть возможность переместить цифру на соответствующее наименованию потомка место пустого поля), а так же значение глубины, на которой располагается элемент |
| class Graph | Node\* root;  Node\* end;  Dataset unsetDataNodes;  List list\_allPath; | Является деревом вариантов, содержит начальное и конечное состояния, хэш-таблица всех вершин, стек вершин |
| unordered\_set<string> Dataset; |  | хэш-таблица всех вершин |
| list<Node\*> List; |  | Стек раскрытых вершин |

**Описание методов класса Node**

|  |  |
| --- | --- |
| Метод | Описание |
| Node(int local\_depth); | Конструктор с параметром глубины |
| void finalPath (Node\* local\_outcome); | Вывод кратчайшего пути нахождения решения. |

**Описание методов класса Graph**

|  |  |
| --- | --- |
| Метод | Описание |
| Graph(); | Инициализация стартовой и целевой вершин графа |
| void DestroyNode(Node\* node); | Уничтожение вершины графа |
| Node\* fillMatrixValues(Node\* local\_node) | Заполнение вершины элементами игрового поля |
| void printMatixNode(Node\* local\_node); | Вывод массива вершин. |
| void printDataNode(Node\* local\_node); | Вывод вершин раскрытых на очередном шаге. |
| void printDataPath(); | Вывод всех вершин содержащихся в кайме. |
| bool check\_repeatNodes(Node\* local\_node) | Проверка на наличие раскрытой вершины в дереве (встречалась ли раньше) |
| bool compareNodes(Node\* local\_node) | Проверка на тождественность слепков состояний игрового поля |
| Node\* fillMatrixValues(Node\* local\_node); | Заполнение массива значениями массива вершины, адрес которой передается в функцию. |
| string getString(Node\* local\_node); | Функция перевода массива вершины в строку для загрузки в unordered\_set и для проверки повторного состояния. |
| void uncoverNodes(Node\* local\_node, int\* local\_nodes\_steps); | Функция для раскрытия вершин потомков от передаваемой вершины предка. |
| bool compareNodes(Node\* local\_node); | Функция сравнения очередной вершины с искомой вершиной для выявления конечного состояния. |
| Node\* end\_or\_again\_or\_else(Node\*\* local\_node\_pointer, int\* local\_nodes\_steps, bool local\_limit\_check, bool local\_steps\_output); | Метод «распознавания» вершины: тождественна искомой/была получена ранее/ другая (уникальная новая) |
| Node\* dfs(int\* local\_nodes\_steps); | Метод поиска в глубину |
| Node\* iterativeDFS(int restriction, int\* local\_nodes\_steps); | Метод поиска в глубину итеративно |
| Node\* dfsBySteps(int\* local\_nodes\_steps); | Метод поиска в глубину по шагам |
| Node\* iterativeDFSBySteps(int restriction, int\* local\_nodes\_steps); | Метод поиска в глубину итеративно по шагам |

**Описание алгоритмов**

**Алгоритм поиска в глубину:**

Поиск в глубину всегда раскрывает одну из вершин на самом глубоком уровне дерева. Останавливается, когда поиск достигает цели или заходит в тупик (ни одна вершина не может быть раскрыта). В последнем случае выполняется возврат назад и раскрываются вершины на более верхних уровнях.

Обход в глубину, в нашем случае, реализуется с помощью стека, который реализован на списке. Алгоритм заключается в следующем: после раскрытия очередной вершины все ее потомки поочерёдно проверяются на соответствие конечному результату или повторному состоянию и помещаются в стек, если не являются таковыми, далее и стека достается верхний элемент и алгоритм повторяется (до того момента пока в стеке либо не останется ни одной вершины, либо не будет найдено искомое состояние).

**Алгоритм с итеративным углублением:**

Тот же поиск в глубину с изначальным пределом глубины, который итеративно увеличивается, если при заданном ранее ограничении результат был не найден.

Реализуется с помощью надстройки над алгоритмом поиска в глубину с ограничением в виде цикла, инкрементирующего ограничивающую глубину дерева при отсутствии результата на текущей итерации.

**Пошаговый поиск:**

В алгоритмах итеративного поиска и «сначала в глубину» в пошаговой реализации после каждого прохода цикла происходит остановка и вывод информации о раскрытых вершинах на данном шаге с помощью функций printDataNode() и printDataPath().

**Пример работы программы:**

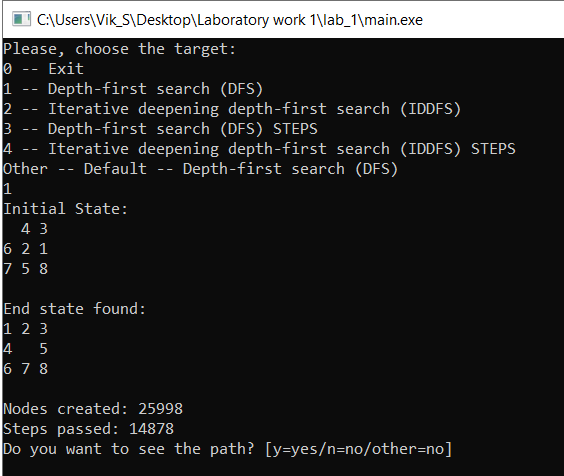


Рис. 1. Поиск в глубину

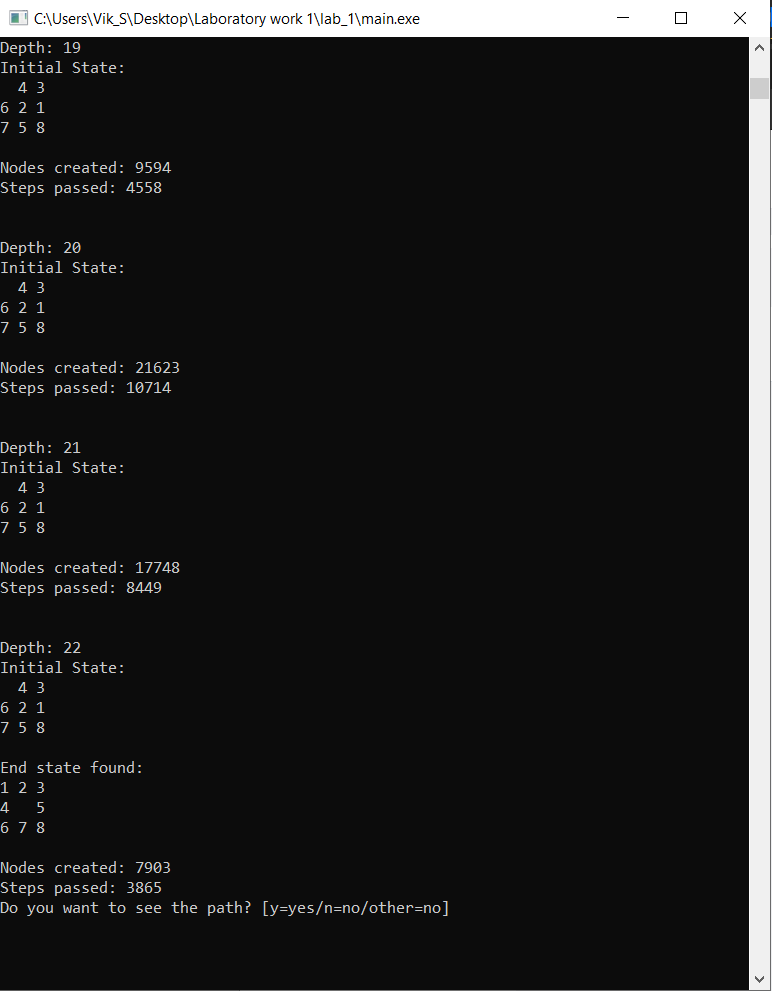


Рис. 2. Итеративный поиск в глубину

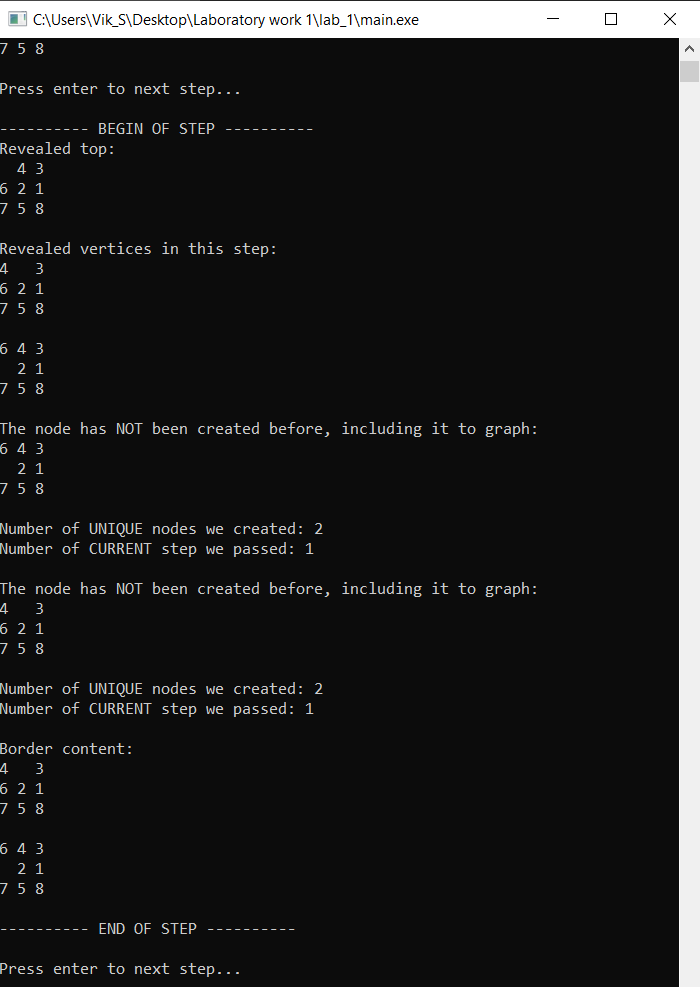


Рис. 3. Поиск в глубину по шагам

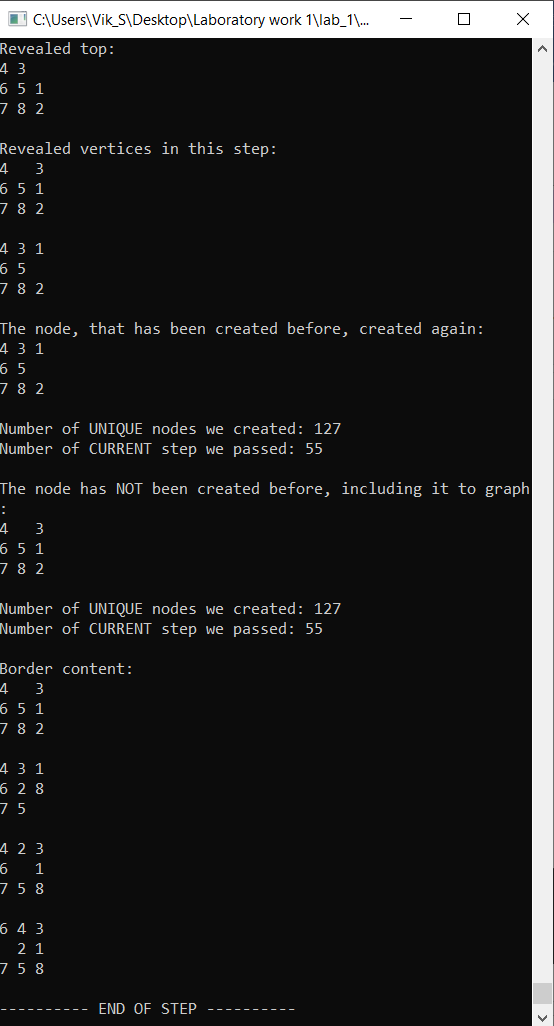


Рис. 4. Итеративный поиск в глубину по шагам

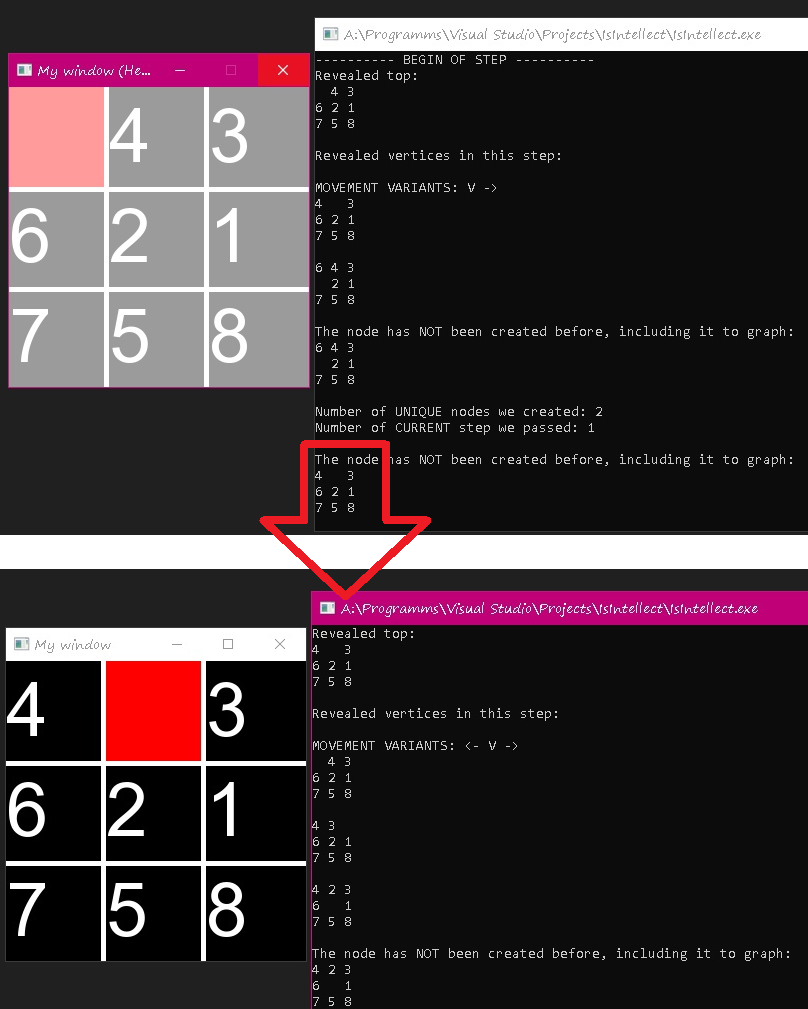
В поздней версии программы была добавлена визуализация шагов:

Рис.4. Визуализация работы алгоритма

**Сравнительные оценки сложности алгоритмов поиска**

|  |  |  |
| --- | --- | --- |
|  | Неинформированный  поиск | |
| Поиск в глубину | Итеративный поиск в глубину |
|
| Временная сложность (кол-во шагов) | 14878 | 3865 |
| Емкостная сложность (кол-во вершин в дереве поиска) | 25998 | 7903 |

Экспериментально доказано, что итеративный поиск имеет преимущества над обычным поиском в глубину как по времени работы, так и по количеству созданных вершин (а значит и расходуемой памяти). Такой результат обусловлен тем, что при итеративном поиске программа не тратит ресурсы на пустое погружение вглубь дерева, в отличии от алгоритма «сначала в глубину», где ввиду невозможности заранее знать глубину искомой вершины, создаётся множество вершин, удлиняющих ветку (например: пока есть возможность – двигать числа вверх), а, имея ограничение по глубине, итеративный поиск является сочетанием свойств алгоритмов поиска в глубину и в ширину – так вероятность дойти до искомой ситуации выше (конкретно для нашей задачи).

**Вывод**

В рамках первой лабораторной работы команда реализовала несколько стратегий слепого поиска в пространстве. Решили проблему выбора структур данных для специфичной работы алгоритма (решено сделать не рекурсивный алгоритм). Сравнительная оценка алгоритмов поиска в глубину и итеративного поиска показала, что компромисс между поиском в глубину и в ширину является лучшим решением, как по временной, так и по емкостной сложности.

**Листинг программы**

**node.h**

using namespace std;

class Node

{

public:

Node(int local\_depth);

char\*\* elements;

int depth;

struct Node\* left;

struct Node\* mid\_left;

struct Node\* mid\_right;

struct Node\* right;

struct Node\* parent;

static void finalPath(Node\*);

};

**node.cpp**

#include <iostream>

#include "./node.h"

Node::Node(int local\_depth)

{

depth = local\_depth;

elements = new char\* [3];

for (int i = 0; i < 3; i++)

{

elements[i] = new char[3];

}

left = nullptr;

mid\_left = nullptr;

mid\_right = nullptr;

right = nullptr;

parent = nullptr;

}

void Node::finalPath (Node\* local\_outcome)

{

int i = 0;

int j = 0;

cout << "Here's the right path from the begin to the solution:\n";

while (local\_outcome != nullptr)

{

for (i = 0; i < 3; i++)

{

for (j = 0; j < 3; j++)

{

cout << local\_outcome->elements[i][j] << " ";

}

cout << "\n";

}

cout << "\n";

local\_outcome = local\_outcome->parent;

}

}

**graph.h**

#include <./node.h>

#include <list>

#include <unordered\_set>

#ifndef GRAPH\_H

#define GRAPH\_H

class Graph

{

public:

Graph();

~Graph();

Node\* dfs(int\* local\_nodes\_steps);

Node\* iterativeDFS(int, int\* local\_nodes\_steps);

Node\* dfsBySteps(int\* local\_nodes\_steps);

Node\* iterativeDFSBySteps(int, int\* local\_nodes\_steps);

Node\* fillMatrixValues(Node\* local\_node);

Node\* end\_or\_again\_or\_else (Node\*\* local\_node\_pointer, int\* local\_nodes\_steps, bool local\_limit\_check, bool local\_steps\_output);

bool compareNodes(Node\* local\_node);

bool check\_repeatNodes(Node\* local\_node);

string getString(Node\* local\_node);

void uncoverNodes(Node\* local\_node, int\* local\_nodes\_steps);

void printDataNode(Node\* local\_node);

void printMatixNode(Node\* local\_node);

void printDataPath();

Node\* root;

Node\* end;

private:

static void DestroyNode(Node\* node);

typedef unordered\_set<string> Dataset;

typedef list<Node\*> List;

Dataset unsetDataNodes;

List list\_allPath;

};

#endif

**graph.cpp**

#include <iostream>

#include "./node.h"

#include "./graph.h"

// ---------- < Default Graph constructor > ----------

Graph::Graph()

{

Node\* startNode = new Node(0);

Node\* endNode = new Node(0);

startNode -> elements[0][0] = ' ';

startNode -> elements[0][1] = '4';

startNode -> elements[0][2] = '3';

startNode -> elements[1][0] = '6';

startNode -> elements[1][1] = '2';

startNode -> elements[1][2] = '1';

startNode -> elements[2][0] = '7';

startNode -> elements[2][1] = '5';

startNode -> elements[2][2] = '8';

// ^^^^^^^^^ //

// ^ - 4 3 ^ //

// ^^^^^^^^^ //

// ^ 6 2 1 ^ //

// ^^^^^^^^^ //

// ^ 7 5 8 ^ //

// ^^^^^^^^^ //

endNode -> elements[0][0] = '1';

endNode -> elements[0][1] = '2';

endNode -> elements[0][2] = '3';

endNode -> elements[1][0] = '4';

endNode -> elements[1][1] = ' ';

endNode -> elements[1][2] = '5';

endNode -> elements[2][0] = '6';

endNode -> elements[2][1] = '7';

endNode -> elements[2][2] = '8';

// ^^^^^^^^^ //

// ^ 1 2 3 ^ //

// ^^^^^^^^^ //

// ^ 4 - 5 ^ //

// ^^^^^^^^^ //

// ^ 6 7 8 ^ //

// ^^^^^^^^^ //

root = startNode;

end = endNode;

}

// ---------- < Default Graph destructor > ----------

Graph::~Graph()

{

DestroyNode(root);

}

// ---------- < Funtion for destroy node of graph > ----------

void Graph::DestroyNode(Node\* node)

{

if (node)

{

DestroyNode(node->left);

DestroyNode(node->mid\_left);

DestroyNode(node->mid\_right);

DestroyNode(node->right);

delete node;

}

}

// --------------------------------------------------

// ---------- <<< ALGORITHMS SEARCHING >>> ----------

// --------------------------------------------------

// ---------- < DFS DEPTH > ----------

Node\* Graph::dfs(int\* local\_nodes\_steps)

{

Node\* local\_node;

Node\* local\_returned\_node = nullptr;

cout << "Initial State:\n";

printMatixNode(root);

unsetDataNodes.insert(getString(root));

list\_allPath.push\_front(root);

local\_node = root;

while (!list\_allPath.empty())

{

local\_node = \*(list\_allPath.begin());

list\_allPath.pop\_front();

this->uncoverNodes(local\_node, local\_nodes\_steps);

local\_nodes\_steps[1] = local\_nodes\_steps[1] + 1;

local\_returned\_node = this->end\_or\_again\_or\_else (&(local\_node->right), local\_nodes\_steps, true, false); // swap possibility/moving possibility

if (local\_returned\_node != nullptr)

{

return local\_returned\_node;

}

local\_returned\_node = this->end\_or\_again\_or\_else (&(local\_node->mid\_right), local\_nodes\_steps, true, false); // swap possibility/moving possibility

if (local\_returned\_node != nullptr)

{

return local\_returned\_node;

}

local\_returned\_node = this->end\_or\_again\_or\_else (&(local\_node->mid\_left), local\_nodes\_steps, true, false); // swap possibility/moving possibility

if (local\_returned\_node != nullptr)

{

return local\_returned\_node;

}

local\_returned\_node = this->end\_or\_again\_or\_else (&(local\_node->left), local\_nodes\_steps, true, false); // swap possibility/moving possibility

if (local\_returned\_node != nullptr)

{

return local\_returned\_node;

}

}

return nullptr;

}

// ---------- < DFS DEPTH LIMIT > ----------

Node\* Graph::iterativeDFS(int restriction, int\* local\_nodes\_steps)

{

Node\* local\_node;

Node\* local\_returned\_node = nullptr;

cout << "Initial State:\n";

printMatixNode(root);

unsetDataNodes.insert(getString(root));

list\_allPath.push\_front(root);

local\_node = root;

while (!list\_allPath.empty())

{

local\_node = \*(list\_allPath.begin());

list\_allPath.pop\_front();

uncoverNodes(local\_node, local\_nodes\_steps);

local\_nodes\_steps[1] = local\_nodes\_steps[1] + 1;

// https://cplusplus.com/doc/tutorial/operators/

local\_returned\_node = this->end\_or\_again\_or\_else (&(local\_node->right), local\_nodes\_steps, local\_node->depth < restriction ? true : false, false); // swap possibility/moving possibility

if (local\_returned\_node != nullptr)

{

return local\_returned\_node;

}

local\_returned\_node = this->end\_or\_again\_or\_else (&(local\_node->mid\_right), local\_nodes\_steps, local\_node->depth < restriction ? true : false, false); // swap possibility/moving possibility

if (local\_returned\_node != nullptr)

{

return local\_returned\_node;

}

local\_returned\_node = this->end\_or\_again\_or\_else (&(local\_node->mid\_left), local\_nodes\_steps, local\_node->depth < restriction ? true : false, false); // swap possibility/moving possibility

if (local\_returned\_node != nullptr)

{

return local\_returned\_node;

}

local\_returned\_node = this->end\_or\_again\_or\_else (&(local\_node->left), local\_nodes\_steps, local\_node->depth < restriction ? true : false, false); // swap possibility/moving possibility

if (local\_returned\_node != nullptr)

{

return local\_returned\_node;

}

}

return nullptr;

}

// ---------- < DFS DEPTH BY STEPS > ----------

Node\* Graph::dfsBySteps(int\* local\_nodes\_steps)

{

Node\* local\_node;

Node\* local\_returned\_node = nullptr;

cout << "Initial State:\n";

printMatixNode(root);

unsetDataNodes.insert(getString(root));

list\_allPath.push\_front(root);

local\_node = root;

printDataPath();

cout << "Press enter to next step...";

fflush(stdin);

getchar();

while (!list\_allPath.empty())

{

local\_node = \*(list\_allPath.begin());

list\_allPath.pop\_front();

cout << "\n---------- BEGIN OF STEP ----------\n";

cout << "Revealed top:\n";

printMatixNode(local\_node);

cout << "Revealed vertices in this step:\n";

uncoverNodes(local\_node, local\_nodes\_steps);

local\_nodes\_steps[1] = local\_nodes\_steps[1] + 1;

printDataNode(local\_node);

local\_returned\_node = this->end\_or\_again\_or\_else (&(local\_node->right), local\_nodes\_steps, true, true); // swap possibility/moving possibility

if (local\_returned\_node != nullptr)

{

return local\_returned\_node;

}

local\_returned\_node = this->end\_or\_again\_or\_else (&(local\_node->mid\_right), local\_nodes\_steps, true, true); // swap possibility/moving possibility

if (local\_returned\_node != nullptr)

{

return local\_returned\_node;

}

local\_returned\_node = this->end\_or\_again\_or\_else (&(local\_node->mid\_left), local\_nodes\_steps, true, true); // swap possibility/moving possibility

if (local\_returned\_node != nullptr)

{

return local\_returned\_node;

}

local\_returned\_node = this->end\_or\_again\_or\_else (&(local\_node->left), local\_nodes\_steps, true, true); // swap possibility/moving possibility

if (local\_returned\_node != nullptr)

{

return local\_returned\_node;

}

printDataPath();

cout << "---------- END OF STEP ----------\n\n";

cout << "Press enter to next step...";

fflush(stdin);

getchar();

}

return nullptr;

}

// ---------- < DFS DEPTH LIMIT BY STEPS > ----------

Node\* Graph::iterativeDFSBySteps(int restriction, int\* local\_nodes\_steps)

{

Node\* local\_node;

Node\* local\_returned\_node = nullptr;

cout << "Initial State:\n";

printMatixNode(root);

unsetDataNodes.insert(getString(root));

list\_allPath.push\_front(root);

local\_node = root;

printDataPath();

cout << "Press enter to next step...";

fflush(stdin);

getchar();

while (!list\_allPath.empty()) {

local\_node = \*(list\_allPath.begin());

list\_allPath.pop\_front();

cout << "\n---------- BEGIN OF STEP ----------\n";

cout << "Revealed top: " << endl;

printMatixNode(local\_node);

cout << "Revealed vertices in this step: " << endl;

uncoverNodes(local\_node, local\_nodes\_steps);

local\_nodes\_steps[1] = local\_nodes\_steps[1] + 1;

printDataNode(local\_node);

local\_returned\_node = this->end\_or\_again\_or\_else (&(local\_node->right), local\_nodes\_steps, local\_node->depth < restriction ? true : false, true); // swap possibility/moving possibility

if (local\_returned\_node != nullptr)

{

return local\_returned\_node;

}

local\_returned\_node = this->end\_or\_again\_or\_else (&(local\_node->mid\_right), local\_nodes\_steps, local\_node->depth < restriction ? true : false, true); // swap possibility/moving possibility

if (local\_returned\_node != nullptr)

{

return local\_returned\_node;

}

local\_returned\_node = this->end\_or\_again\_or\_else (&(local\_node->mid\_left), local\_nodes\_steps, local\_node->depth < restriction ? true : false, true); // swap possibility/moving possibility

if (local\_returned\_node != nullptr)

{

return local\_returned\_node;

}

local\_returned\_node = this->end\_or\_again\_or\_else (&(local\_node->left), local\_nodes\_steps, local\_node->depth < restriction ? true : false, true); // swap possibility/moving possibility

if (local\_returned\_node != nullptr)

{

return local\_returned\_node;

}

printDataPath();

cout << "---------- END OF STEP ----------\n\n";

cout << "Press enter to next step...";

fflush(stdin);

getchar();

}

return nullptr;

}

// --------------------------------------------------

// ---------- <<< OTHER FUNTIONS >>> ----------

// --------------------------------------------------

// ---------- < REPEAT NODES > ----------

bool Graph::check\_repeatNodes(Node\* local\_node) {

if (unsetDataNodes.find(getString(local\_node)) == unsetDataNodes.end()) {

return true;

}

else {

return false;

}

}

// ---------- < COMPARE NODES > ----------

bool Graph::compareNodes(Node\* local\_node) {

for (int i = 0; i < 3; i++) {

for (int j = 0; j < 3; j++) {

for (int y = 0; y < 3; y++) {

for (int x = 0; x < 3; x++) {

if (local\_node->elements[i][j] == end->elements[y][x]) {

if (i != y || j != x) {

return false;

}

}

}

}

}

}

return true;

}

// ---------- < PRINT DATA NODE> ----------

void Graph::printDataNode(Node\* local\_node) {

if (local\_node->left != nullptr) {

printMatixNode(local\_node->left);

}

if (local\_node->mid\_left != nullptr) {

printMatixNode(local\_node->mid\_left);

}

if (local\_node->mid\_right != nullptr) {

printMatixNode(local\_node->mid\_right);

}

if (local\_node->right != nullptr) {

printMatixNode(local\_node->right);

}

}

// ---------- < PRINT ALL SUCCESS END STATES TO FINISH PATH > ----------

void Graph::printDataPath() {

List::iterator iter;

cout << "Border content: " << endl;

for (iter = list\_allPath.begin(); iter != list\_allPath.end(); iter++) {

printMatixNode(\*iter);

}

}

// ---------- < PRINT ONLY NODE MATRIX > ----------

void Graph::printMatixNode(Node\* local\_node) {

for (int i = 0; i < 3; i++) {

for (int j = 0; j < 3; j++) {

cout << local\_node->elements[i][j] << " ";

}

cout << endl;

}

cout << endl;

}

// ---------- < GETSTRING > ----------

string Graph::getString(Node\* local\_node)

{

string str;

for (int i = 0; i < 3; i++)

{

for (int j = 0; j < 3; j++)

{

str.push\_back((\*local\_node).elements[i][j]);

}

}

return str;

}

// ---------- < Uncover Nodes > ----------

void Graph::uncoverNodes(Node\* local\_node, int\* local\_nodes\_steps)

{

int x, y;

for (int i = 0; i < 3; i++) {

for (int j = 0; j < 3; j++) {

if ((\*local\_node).elements[i][j] == ' ') {

x = j;

y = i;

break;

}

}

}

if (x > 0) {

(\*local\_node).left = fillMatrixValues(local\_node);

(\*local\_node).left->elements[y][x] = (\*local\_node).left->elements[y][x - 1];

(\*local\_node).left->elements[y][x - 1] = ' ';

local\_nodes\_steps[0] = local\_nodes\_steps[0] + 1;

(\*local\_node).left->parent = local\_node;

}

if (x < 2) {

(\*local\_node).mid\_right = fillMatrixValues(local\_node);

(\*local\_node).mid\_right->elements[y][x] = (\*local\_node).mid\_right->elements[y][x + 1];

(\*local\_node).mid\_right->elements[y][x + 1] = ' ';

local\_nodes\_steps[0] = local\_nodes\_steps[0] + 1;

(\*local\_node).mid\_right->parent = local\_node;

}

if (y > 0) {

(\*local\_node).mid\_left = fillMatrixValues(local\_node);

(\*local\_node).mid\_left->elements[y][x] = (\*local\_node).mid\_left->elements[y - 1][x];

(\*local\_node).mid\_left->elements[y - 1][x] = ' ';

local\_nodes\_steps[0] = local\_nodes\_steps[0] + 1;

(\*local\_node).mid\_left->parent = local\_node;

}

if (y < 2) {

(\*local\_node).right = fillMatrixValues(local\_node);

(\*local\_node).right->elements[y][x] = (\*local\_node).right->elements[y + 1][x];

(\*local\_node).right->elements[y + 1][x] = ' ';

local\_nodes\_steps[0] = local\_nodes\_steps[0] + 1;

(\*local\_node).right->parent = local\_node;

}

}

// ---------- < FILL MATRIX VALUES > ----------

Node\* Graph::fillMatrixValues(Node\* local\_node)

{

Node\* NewNode = new Node(local\_node->depth + 1);

for (int i = 0; i < 3; i++) {

for (int j = 0; j < 3; j++) {

(\*NewNode).elements[i][j] = (\*local\_node).elements[i][j];

}

}

return NewNode;

}

// ---------- < COMPARING IF IT'S END, OR IF IT'S EXISTING NODE, OR IF IT'S NEW, BUT NOT THE END > ----------

Node\* Graph::end\_or\_again\_or\_else (Node\*\* local\_node\_pointer, int\* local\_nodes\_steps, bool local\_limit\_check, bool local\_steps\_output)

{

if (\*local\_node\_pointer != nullptr && local\_limit\_check == true)

{

if (this->check\_repeatNodes(\*local\_node\_pointer))

{

if (this->compareNodes(\*local\_node\_pointer))

{

cout << "End state found:\n";

this->printMatixNode(\*local\_node\_pointer);

return \*local\_node\_pointer;

}

if (local\_steps\_output == true)

{

cout << "The node has NOT been created before, including it to graph:\n";

printMatixNode(\*local\_node\_pointer);

}

this->unsetDataNodes.insert(getString(\*local\_node\_pointer));

this->list\_allPath.push\_front(\*local\_node\_pointer);

if (local\_steps\_output == true)

{

cout << "Number of UNIQUE nodes we created: " << local\_nodes\_steps[0] << "\n"; // UNIQUE means we don't count any of similar nodes

cout << "Number of CURRENT step we passed: " << local\_nodes\_steps[1] << "\n\n";

}

return nullptr;

}

else

{

if (local\_steps\_output == true)

{

cout << "The node, that has been created before, created again:\n";

printMatixNode(\*local\_node\_pointer);

}

delete(\*local\_node\_pointer);

\*local\_node\_pointer = nullptr;

local\_nodes\_steps[0] = local\_nodes\_steps[0] - 1;

if (local\_steps\_output == true)

{

cout << "Number of UNIQUE nodes we created: " << local\_nodes\_steps[0] << "\n"; // UNIQUE means we don't count any of similar nodes

cout << "Number of CURRENT step we passed: " << local\_nodes\_steps[1] << "\n\n";

}

return \*local\_node\_pointer;

}

}

return nullptr;

}

**main.cpp**

#include <iostream>

#include <stack>

#include <unordered\_set>

#include <string>

#include <list>

#include "./node.h"

#include "./graph.h"

using namespace std;

int main()

{

// ---------- BEGIN ----------

// here we initializing variables, including main graph class & outcome node,

// number of nodes created & steps passed.

// also we get choosed algorithm option by user

int algorithm\_option = 0;

int created\_nodes\_steps\_passed[2];

Graph\* graph = nullptr; // https://stackoverflow.com/questions/18940175/cannot-delete-stdvector-stdarray

Node\* outcome = nullptr;

cout << "Please, choose the target:\n"

<< "0 -- Exit\n"

<< "1 -- Depth-first search (DFS)\n"

<< "2 -- Iterative deepening depth-first search (IDDFS)\n"

<< "3 -- Depth-first search (DFS) STEPS\n"

<< "4 -- Iterative deepening depth-first search (IDDFS) STEPS\n"

<< "Other -- Default -- Depth-first search (DFS)\n";

cin >> algorithm\_option;

created\_nodes\_steps\_passed[0] = 0;

created\_nodes\_steps\_passed[1] = 0;

graph = new Graph();

// ---------- ALGORITHM ----------

// here we doing the choosed algorithm option & showing created nodes & passed steps

switch (algorithm\_option) // https://stackoverflow.com/questions/34829955/what-is-causing-this-cannot-jump-from-switch-statement-to-this-case-label

{

case 0: // nothing, exit

{

break;

}

case 1: // DFS

{

outcome = graph->dfs(created\_nodes\_steps\_passed);

cout << "Nodes created: " << created\_nodes\_steps\_passed[0] << "\n";

cout << "Steps passed: " << created\_nodes\_steps\_passed[1] << "\n";

break;

}

case 2: // idDFS

{

int local\_number = 0;

while (outcome == nullptr)

{

delete graph;

graph = new Graph();

cout << "\n\nDepth: " << local\_number << "\n";

outcome = graph->iterativeDFS(local\_number, created\_nodes\_steps\_passed);

local\_number = local\_number + 1;

cout << "Nodes created: " << created\_nodes\_steps\_passed[0] << "\n";

cout << "Steps passed: " << created\_nodes\_steps\_passed[1] << "\n";

created\_nodes\_steps\_passed[0] = 0;

created\_nodes\_steps\_passed[1] = 0;

}

break;

}

case 3: // DFS steps

{

outcome = graph->dfsBySteps(created\_nodes\_steps\_passed);

cout << "Nodes created: " << created\_nodes\_steps\_passed[0] << "\n";

cout << "Steps passed: " << created\_nodes\_steps\_passed[1] << "\n";

break;

}

case 4: // idDFS steps

{

int local\_number = 0;

while (outcome == nullptr)

{

delete graph;

graph = new Graph();

cout << "\n\nDepth: " << local\_number << "\n";

outcome = graph->iterativeDFSBySteps(local\_number, created\_nodes\_steps\_passed);

local\_number = local\_number + 1;

cout << "Nodes created: " << created\_nodes\_steps\_passed[0] << "\n";

cout << "Steps passed: " << created\_nodes\_steps\_passed[1] << "\n";

created\_nodes\_steps\_passed[0] = 0;

created\_nodes\_steps\_passed[1] = 0;

}

break;

}

default: // default DFS

{

outcome = graph->dfs(created\_nodes\_steps\_passed);

cout << "Nodes created: " << created\_nodes\_steps\_passed[0] << "\n";

cout << "Steps passed: " << created\_nodes\_steps\_passed[1] << "\n";

break;

}

}

// ---------- FOUND PATH OUTPUT / NO FOUND PATH MESSAGE ----------

// if there is path, we ask user if he wants to see it (yes or no),

// if he wants (y), we show it,

// if he doesn't want (n/other), we don't show it.

if (outcome != nullptr) // if outcome is not nullptr, print

{

char local\_path = 'n';

cout << "Do you want to see the path? [y=yes/n=no/other=no]";

fflush(stdin);

cin >> local\_path;

if (local\_path == 'y')

{

Node::finalPath(outcome);

}

}

else

{

cout << "The path hasn't been found.\n";

}

// ---------- END ----------

// here we are freeing the nodes allocated memory

// (the nodes pointers are not freed because the has been

// freed with graph pointer, it has it's own destructor)

// and waiting user to press "enter" or something else

// to sontinue and finish

delete graph;

cout << "Press enter to exit...";

fflush(stdin);

getchar();

}

**makefile**

main: main.cpp node.cpp graph.cpp

g++ -o main main.cpp node.cpp graph.cpp -I.