**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра Вычислительной техники**

отчЁт

**по лабораторной работе №2**

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

**Тема: «МЕТОДЫ ИНФОРМИРОВАННОГО (ЭВРИСТИЧЕСКОГО) ПОИСКА»**

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

Санкт-Петербург

2022г.

**Цель работы**

Практическое закрепление теоретических основ информированного (эвристического) поиска.

**Постановка задачи**

Реализовать программу поиска пути решения головоломки «8-ка» с использованием эвристических функций h1 (поиск фишек не на своих местах с присваиванием приоритета) и h2 (манхэттенское расстояние) для алгоритма поиска A\* 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.

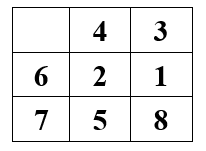


Рис. 1. Начальное состояние

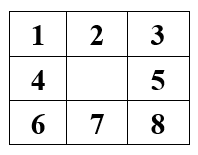


Рис. 2. Конечное состояние

**Распределение обязанностей**

Аюпов Ренат – модификация алгоритма слепого поиска, реализация алгоритма поиска A\*, реализация эвристической функции h1 (поиск фишек не на своих местах с присваиванием приоритета), обновление и добавление зависимостей структур данных.

Соболев Матвей – модификация алгоритма слепого поиска, реализация эвристической функции h2 (манхэттенское расстояние), обновление и добавление структур данных для задания лабораторной работы 2.

Степовик Виктор – сравнительная оценка временной сложности, описание классов, тестирование и отладка программы, доработка архитектуры программы для задания лабораторной работы 2.

**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\* up;  struct Node\* down;  struct Node\* right;  struct Node\* parent;  int priority; | 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 также значение глубины, на которой располагается элемент, и значение приоритета, который показывает либо количество фишек, находящихся не на своих местах, либо сумму модулей разности координат фишек текущей и конечной вершин. |
| class Graph | Node\* root;  Node\* end;  Dataset unsetDataNodes;  List list\_allPath;  Queue\_p priority\_queue; | Является деревом вариантов, содержит начальное и конечное состояния, хэш-таблицу всех вершин, стек вершин, очередь с приоритетом. |
| struct comparator |  | Перегружающая структура, предназначенная для сравнения приоритета вершин. |
| unordered\_set<string> Dataset; |  | Хэш-таблица всех вершин. |
| list<Node\*> List; |  | Стек раскрытых вершин. |
| priority\_queue<Node\*, std::vector<Node\*>, comparator> Queue\_p; |  | Очередь с приоритетом. |

**Описание методов класса Node**

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

**Описание методов класса Graph**

|  |  |
| --- | --- |
| Метод | Описание |
| Graph(); | Инициализация стартовой и целевой вершин графа. |
| static 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. |
| void choiceOfHeuristic(Node\* local\_node, int algorithm\_settingsOption); | Меню выбора эвристической функции. |
| int compareH1(Node\* local\_node); | Определение приоритета как количества фишек, находящихся не на своих местах. |
| int compareH2(Node\* local\_node); | Определение приоритета как суммы модулей разности координат текущей и конечной вершин. |
| void f\_h1(Node\* local\_node); | Определение приоритета для жадного поиска эвристики h1. |
| void f\_h2(Node\* local\_node); | Определение приоритета для жадного поиска эвристики h2. |
| int compareAH1(Node\* local\_node); | Определение приоритета как количества фишек, находящихся не на своих местах, плюс глубина. |
| int compareAH2(Node\* local\_node); | Определение приоритета как суммы модулей разности координат текущей и конечной вершин, плюс глубина. |
| void f\_Ah1(Node\* local\_node); | Определение приоритета для алгоритма A\* для эвристики h1. |
| void f\_Ah2(Node\* local\_node); | Определение приоритета для алгоритма A\* для эвристики h2. |

**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().

**Алгоритм A\* (A star)**

Суть алгоритма A\* (A star, A со звёздочкой) заключается в том, что это такой алгоритм поиска, который совмещает в себе поиск по наилучшему совпадению и жадный поиск, но для которого необходима оценочная стоимость.

Жадный поиск, который стремится минимизировать оценочную стоимость до цели , что позволяет в ряде случаев повысить эффективность поиска, однако жадный поиск не является ни оптимальным, ни полным. С другой стороны, поиск по стоимости или по критерию стоимости минимизирует стоимость пути до текущего состояния и является полным и оптимальным. Однако это часто оказывается неэффективным, но ввод аддитивной оценочной стоимости, то есть совмещение 2-х подходов или стратегий, позволил нам использовать их преимущества сразу:

,

где – это значение, посчитанное на основе эвристик, – это значение глубины, на котором находится вершина, а – это вершина в алгоритме.

**Эвристическая функция h1 (поиск фишек не на своих местах с присваиванием приоритета)**

Шаг 1. Выбирается вершина с наименьшей суммой фишек, находящихся не на своих местах.

Шаг 2. Раскрытие потомков вершины с расчётом их приоритета.

Шаг 3. Проверка потомков вершины на соответствие их начальному варианту.

Шаг 4. Проверка на повторное значение и добавление в очередь с приоритетом.

Шаг 5. Ожидание своей очереди согласно их приоритету.

**Эвристическая функция h2 (манхэттенское расстояние)**

Шаги эвристической функции h2 (манхэттенское расстояние) аналогичны эвристической функции h1 за исключением способа определения приоритета вершины – он определяется как сумма модулей разности координат фишек текущей и конечной вершин.

**Пример работы программы**

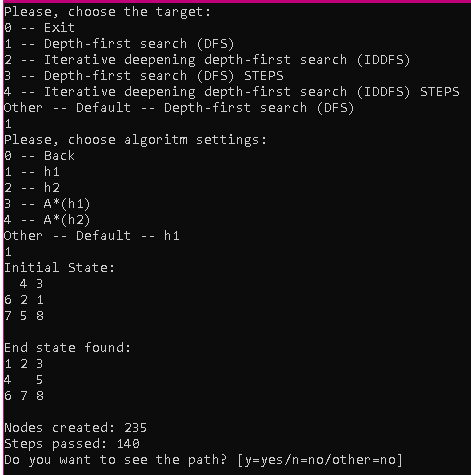


Рисунок 1. Поиск в глубину, эвристика h1

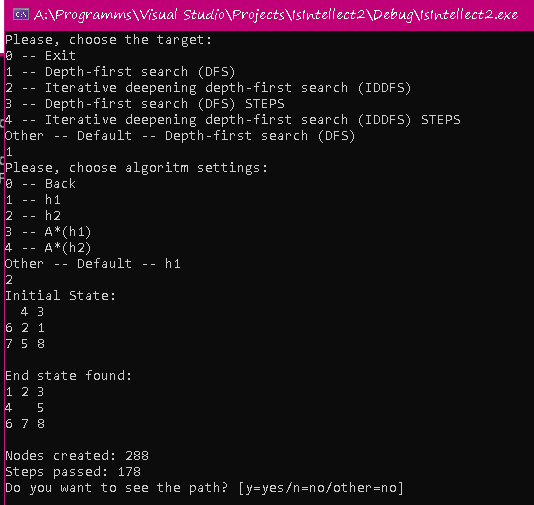


Рисунок 2. Поиск в глубину, эвристика h2

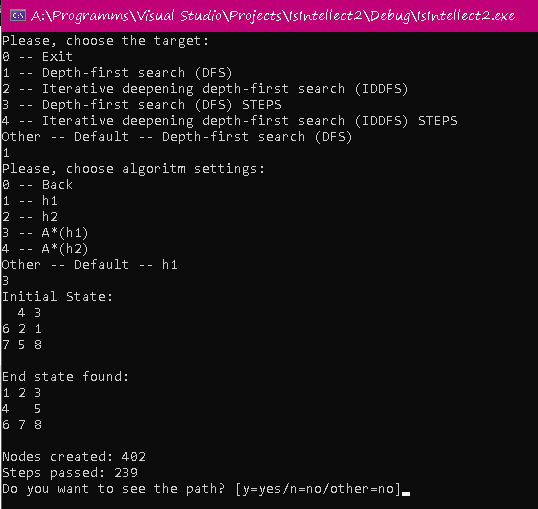


Рисунок 3. Поиск в глубину, алгоритм A\*, эвристика h1

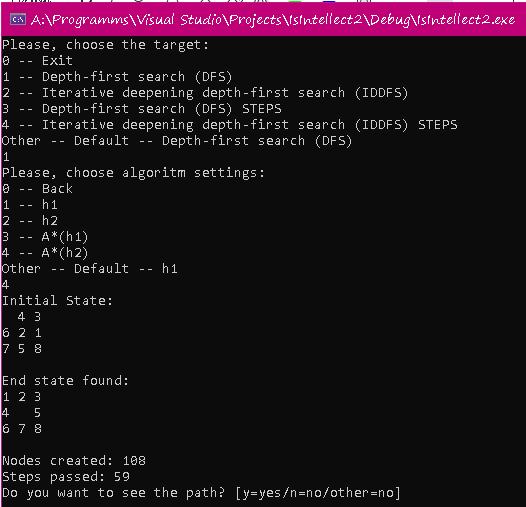


Рисунок 4. Поиск в глубину, алгоритм A\*, эвристика h2

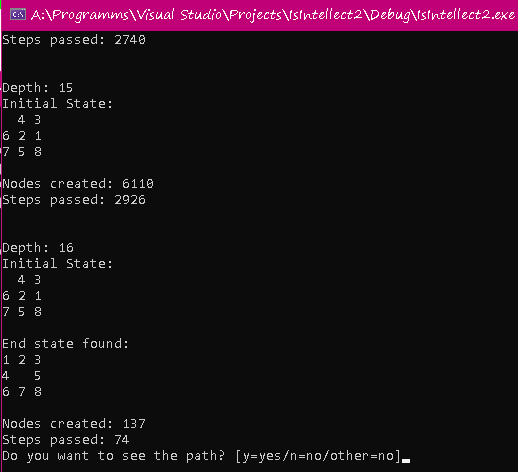
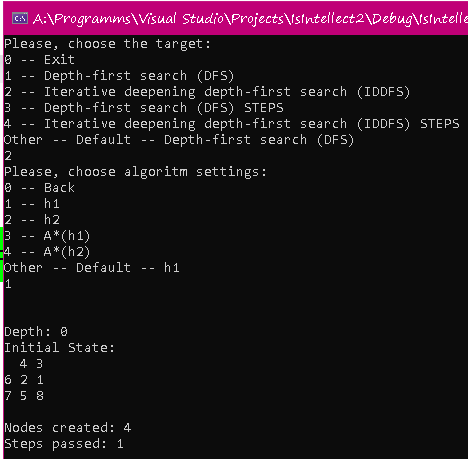


Рисунок 5. Итеративно углубляющийся поиск в глубину, эвристика h1

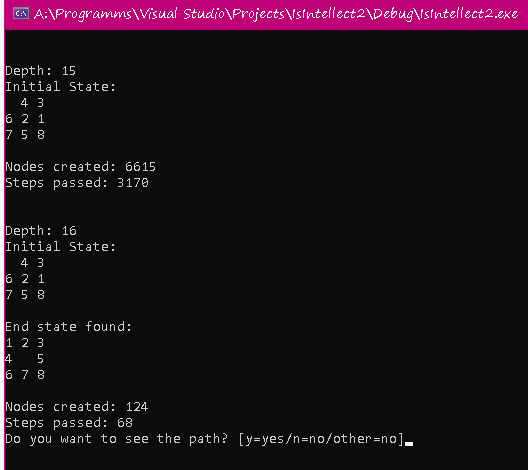
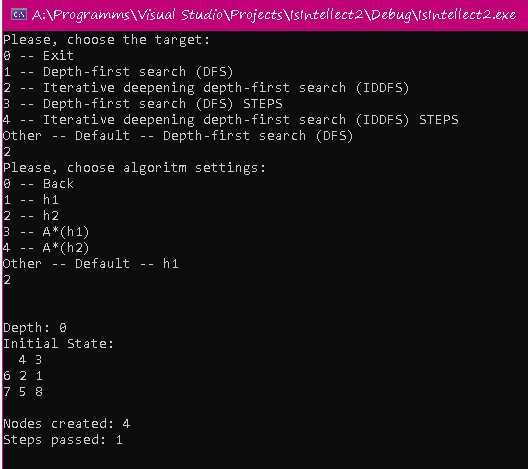


Рисунок 6. Итеративно углубляющийся поиск в глубину, эвристика h2

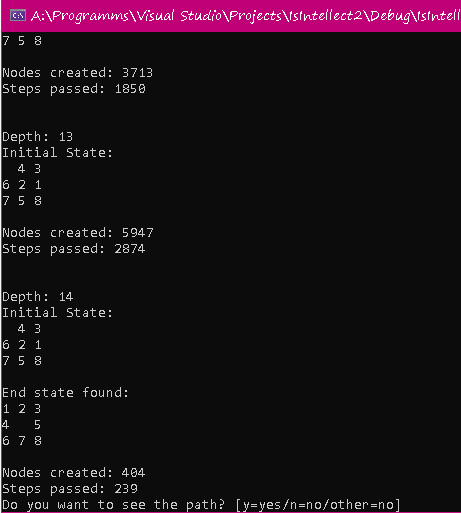
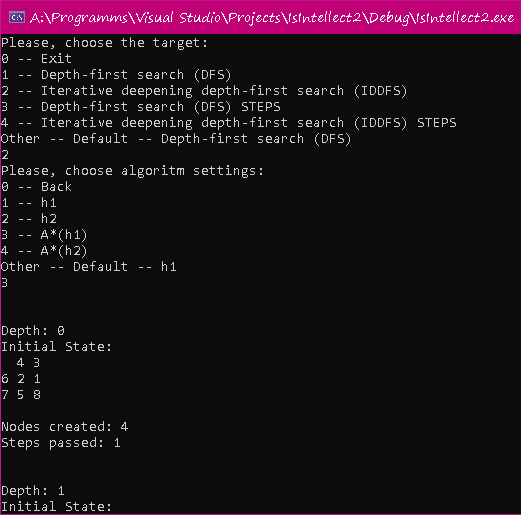


Рисунок 7. Итеративно углубляющийся поиск в глубину, алгоритм A\*, эвристика h1

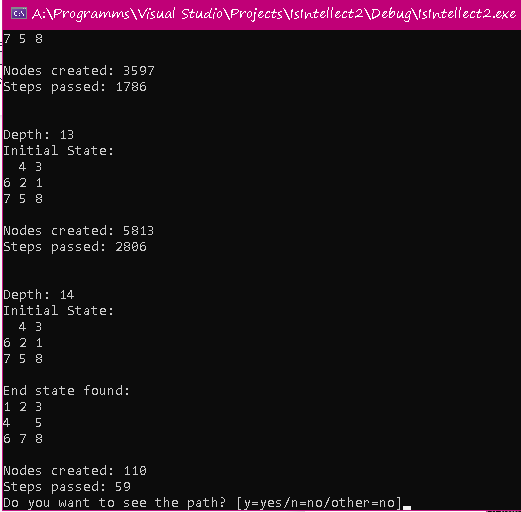
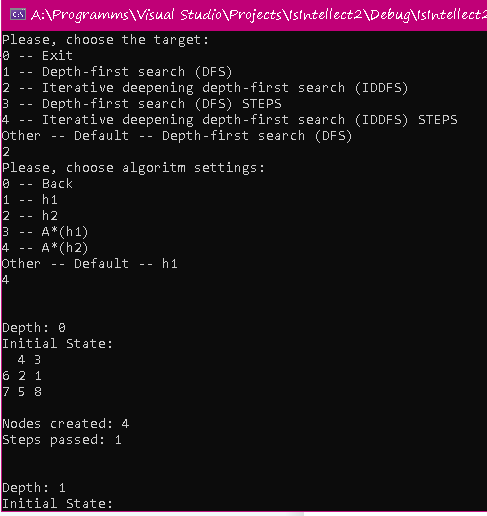
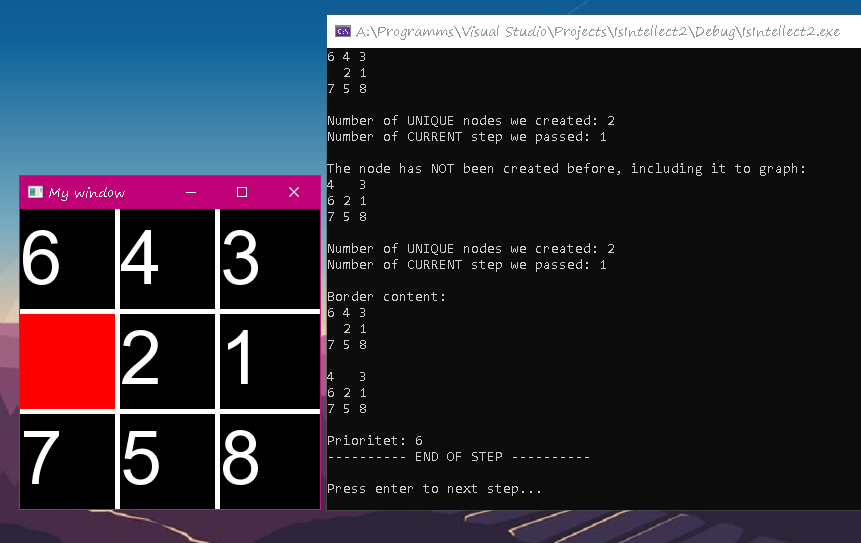
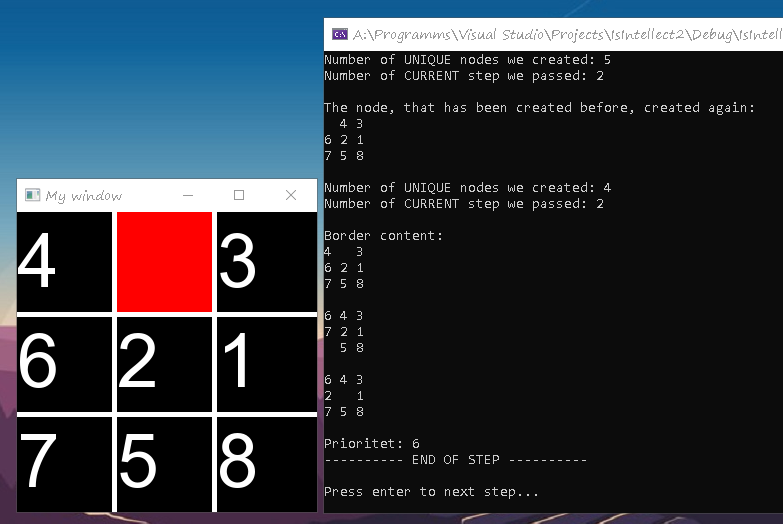


Рисунок 8. Итеративно углубляющийся поиск в глубину, алгоритм A\*, эвристика h2





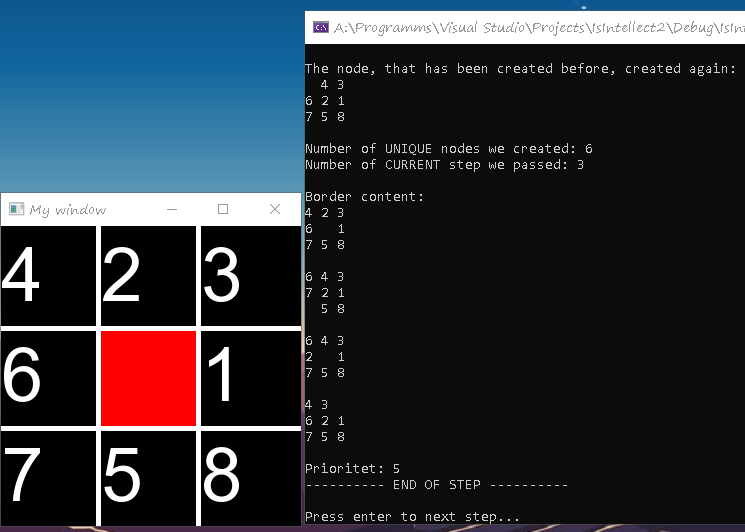
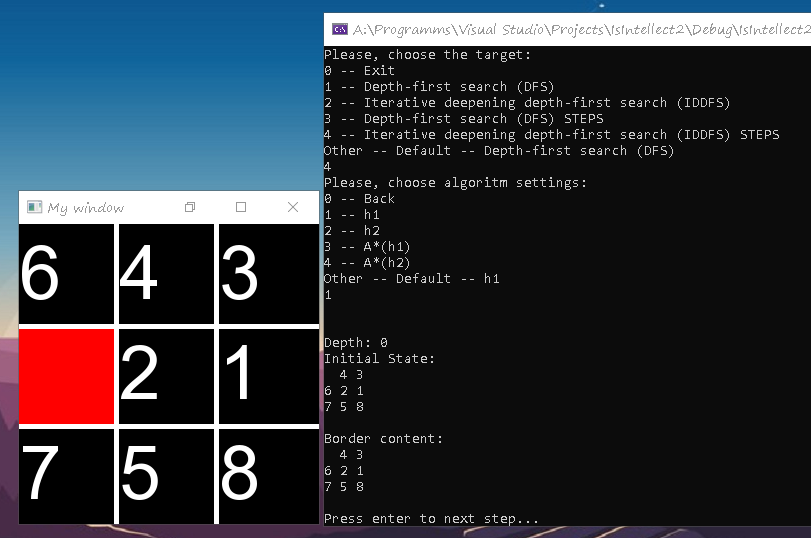


Рисунок 9. Поиск в глубину по шагам, эвристика h1





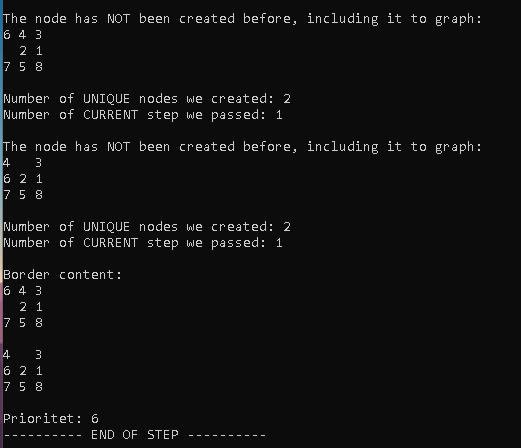


Рисунок 10. Итеративно углубляющийся поиск в глубину по шагам, эвристика h2

**Сравнительные оценки сложности алгоритмов поиска**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Неинформированный  поиск | | Информированный поиск | | | |
| Поиск в глубину | Итеративный поиск в глубину | Жадный поиск (поиск в глубину) | | Алгоритм A\* (поиск в глубину) | |
|  |  |  | h1 | h2 | g + h1 | g + h2 |
| Временная сложность (кол-во шагов) | 14878 | 3865 | 140 | 178 | 239 | 59 |
| Емкостная сложность (кол-во вершин в дереве поиска) | 25998 | 7903 | 235 | 288 | 402 | 108 |

Были проведены эксперименты, которые показывают временную и емкостную сложность алгоритмов и стратегий поиска. По полученным данным информированный жадный поиск для алгоритма поиска в глубину примерно в 100 (h1) или в 80 (h2) раз эффективнее по времени, чем обычный поиск в глубину, а также примерно в 110 (h1) или в 90 (h2) раз эффективнее по ёмкости.

Это объясняется тем, что при неинформированном поиске мы перебираем все вершины (либо все вершины, но с некоторым заданным ограничением, как, например, при итеративно углубляющемся поиске в глубину), пока просто не найдём конечную, а при информированном поиске мы используем информацию в эвристической функции, которая при каждом переборе, то есть раскрытии вершины, оценивает имеющиеся альтернативы (в каком направлении далее раскрывать вершины).

Таким образом, можно заключить, что информированный поиск работает лучше, чем неинформированный, так как он (информированный) оптимизирован и обладает значительным преимуществом: наличием информации о тех или иных путях раскрытия, исходя из которой он может определить наиболее выгодное раскрытие и, как следствие, может улучшить (сократить) путь к конечной вершине.

**Вывод**

В ходе выполнения лабораторной работы №2 «Методы информированного (эвристического) поиска» были реализованы Алгоритм A\*, а также две эвристические функции h1 (поиск фишек не на своих местах с присваиванием приоритета) и h2 (манхэттенское расстояние), для чего были добавлены дополнительные структуры данных, такие как компаратор и очередь с приоритетом вместо стека, далее была сделана временная оценка сложности алгоритма и стратегий информированного поиска, из чего была подтверждена его эффективность и большая эффективность по сравнению с неинформированным поиском. Таким образом и была выполнена лабораторная работа.

В ходе работы возникли следующие проблемы:

1. Нам был необходим компаратор для сравнения приоритета у вершин. Для реализации компаратора можно было бы выбрать 3 варианта: основываясь на лямбда-функциях, основываясь на классе или основываясь на структуре. Была произведена попытка написать его на лямбда-функциях, но при компиляции и написании возникли ошибки, связанные с системными зависимостями, иными словами, сама архитектура программы не позволила использовать лямбда-функции. Поэтому мы решили использовать структуру, так как её было легче всего объявить, описать и интегрировать в уже имеющийся код.
2. В ходе работы возникла проблема выбора типа данных, который должен поддерживать два типа операций: добавить элемент и извлечь максимум. Выбор пал на очередь с приоритетом («priority\_queue»), где элемент с более высоким приоритетом располагается перед элементом с более низким приоритетом, а если у них он одинаковый, они располагаются исходя из изначальной позиции в структуре. Это было нужно для выполнения алгоритма более оптимальным методом, чтобы сразу основываться на очереди, делегируя сортировку объектов структуре данных.

**Листинг программы**

**node.h**

#include <iostream>

#include <unordered\_set>

#include <list>

#include <SFML\Graphics.hpp>

#include <queue>

#ifndef NODE\_H

#define NODE\_H

using namespace std;

struct Node

{

public:

Node(int local\_depth);

char\*\* elements;

int depth;

struct Node\* left;

struct Node\* up;

struct Node\* down;

struct Node\* right;

struct Node\* parent;

static void finalPath(Node\*);

int priority;

};

#endif

**node.cpp**

#include <iostream>

#include "./node.h"

#include <SFML\System\String.hpp>

#include <SFML\Graphics\RenderWindow.hpp>

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;

up = nullptr;

down = nullptr;

right = nullptr;

parent = nullptr;

priority = 0;

}

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 <list>

#include <unordered\_set>

#include <queue>

#include <vector>

#include <SFML\Graphics.hpp>

#include <SFML\Graphics\Font.hpp>

#include <SFML\Graphics\Text.hpp>

#include <SFML\Graphics\RenderWindow.hpp>

struct comparator {

public:

bool operator()(const Node\* a, const Node\* b) // overloading both operators

{

return a->priority > b->priority;

}

};

class Graph

{

protected:

typedef priority\_queue<Node\*, std::vector<Node\*>, comparator > Queue\_p;

public:

/\*----------------------------------------CONSTRUCTOR AND DESTRUCTOR-----------------------------------\*/

Graph();

~Graph();

/\*----------------------------------------CONSTRUCTOR AND DESTRUCTOR-----------------------------------\*/

/\*----------------------------------------ALGORITHMS-----------------------------------\*/

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

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

/\*----------------------------------------ALGORITHMS-----------------------------------\*/

/\*----------------------------------------ALGORITHMS STEPS-----------------------------------\*/

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

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

/\*----------------------------------------ALGORITHMS STEPS-----------------------------------\*/

/\*----------------------------------------FILLING VALUES AND WORK WITH NODES-----------------------------------\*/

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, bool local\_steps\_output, int choose);

void printDataNode(Node\* local\_node);

void printMatixNode(Node\* local\_node);

void printDataPath(Queue\_p queue\_prioritet);

void choiceOfHeuristic(Node\* local\_node, int algorithm\_settingsOption);

int compareH1(Node\* local\_node);

int compareH2(Node\* local\_node);

void f\_h1(Node\* local\_node);

void f\_h2(Node\* local\_node);

int compareAH1(Node\* local\_node);

int compareAH2(Node\* local\_node);

void f\_Ah1(Node\* local\_node);

void f\_Ah2(Node\* local\_node);

/\*----------------------------------------FILLING VALUES AND WORK WITH NODES-----------------------------------\*/

/\*----------------------------------------RENDERING-----------------------------------\*/

void RenderLoad(Node\* local\_node);

void RenderUpdate(Node\* local\_node);

void RenderDraw(sf::RenderWindow&);

/\*----------------------------------------RENDERING-----------------------------------\*/

Node\* root;

Node\* end;

private:

static void DestroyNode(Node\* node);

typedef unordered\_set<string> Dataset;

typedef list<Node\*> List;

Dataset unsetDataNodes;

List list\_allPath;

Queue\_p priority\_queue;

sf::Text cell\_1, cell\_2, cell\_3;

sf::Text cell\_4, cell\_5, cell\_6;

sf::Text cell\_7, cell\_8, cell\_9;

sf::RectangleShape cell\_1s, cell\_2s, cell\_3s;

sf::RectangleShape cell\_4s, cell\_5s, cell\_6s;

sf::RectangleShape cell\_7s, cell\_8s, cell\_9s;

sf::RectangleShape line1, line2, line3, line4;

sf::Font font;

};

**graph.cpp**

#include <iostream>

#include "./node.h"

#include "./graph.h"

#include <SFML/Graphics.hpp>

using namespace sf;

// ---------- < 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->up);

DestroyNode(node->down);

DestroyNode(node->right);

delete node;

}

}

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

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

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

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

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

{

Node\* local\_node = new Node(0);

Node\* local\_returned\_node = nullptr;

cout << "Initial State:\n";

printMatixNode(root);

unsetDataNodes.insert(getString(root));

priority\_queue.push(root);

//list\_allPath.push\_front(root);

local\_node = root;

/\*----------------------------------------------RENDERING------------------------------------------------\*/

RenderWindow window(VideoMode(300, 300), "My window");

this->RenderLoad(local\_node);

/\*----------------------------------------------RENDERING------------------------------------------------\*/

while (!priority\_queue.empty())

{

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

local\_node = priority\_queue.top();

priority\_queue.pop();

//list\_allPath.pop\_front();

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

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->down), 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->up), 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;

}

/\*----------------------------------------------RENDERING------------------------------------------------\*/

if (local\_node != nullptr) {

Event event;

while (window.pollEvent(event))

{

if (event.type == Event::Closed)

window.close();

}

this->RenderUpdate(local\_node);

window.clear(Color::Yellow);

this->RenderDraw(window);

window.display();

}

/\*----------------------------------------------RENDERING------------------------------------------------\*/

}

return nullptr;

}

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

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

{

Node\* local\_node = new Node(0);

Node\* local\_returned\_node = nullptr;

cout << "Initial State:\n";

printMatixNode(root);

unsetDataNodes.insert(getString(root));

//list\_allPath.push\_front(root);

priority\_queue.push(root);

local\_node = root;

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

/\*----------------------------------------------RENDERING------------------------------------------------\*/

RenderWindow window(VideoMode(300, 300), "My window");

this->RenderLoad(local\_node);

/\*----------------------------------------------RENDERING------------------------------------------------\*/

while (!priority\_queue.empty())

{

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

local\_node = priority\_queue.top();

priority\_queue.pop();

//list\_allPath.pop\_front();

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

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

/\*----------------------------------------------RENDERING------------------------------------------------\*/

if (local\_node != nullptr) {

Event event;

while (window.pollEvent(event))

{

if (event.type == Event::Closed)

window.close();

}

this->RenderUpdate(local\_node);

window.clear(Color::Yellow);

this->RenderDraw(window);

window.display();

}

/\*----------------------------------------------RENDERING------------------------------------------------\*/

// 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->down), 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->up), 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, int choose)

{

Node\* local\_node = new Node(0);

Node\* local\_returned\_node = nullptr;

cout << "Initial State:\n";

printMatixNode(root);

unsetDataNodes.insert(getString(root));

priority\_queue.push(root);

list\_allPath.push\_front(root);

local\_node = root;

printDataPath(priority\_queue);

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

fflush(stdin);

getchar();

/\*----------------------------------------------RENDERING------------------------------------------------\*/

RenderWindow window(VideoMode(300, 300), "My window");

this->RenderLoad(local\_node);

/\*----------------------------------------------RENDERING------------------------------------------------\*/

while (!priority\_queue.empty())

{

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

local\_node = priority\_queue.top();

priority\_queue.pop();

//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, true, choose);

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->down), 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->up), 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(priority\_queue);

local\_node = priority\_queue.top();

cout << "Prioritet: " << local\_node->priority << endl;

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

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

/\*----------------------------------------------RENDERING------------------------------------------------\*/

if (local\_node != nullptr) {

Event event;

while (window.pollEvent(event))

{

if (event.type == Event::Closed)

window.close();

}

this->RenderUpdate(local\_node);

window.clear(Color::Yellow);

this->RenderDraw(window);

window.display();

}

/\*----------------------------------------------RENDERING------------------------------------------------\*/

fflush(stdin);

getchar();

}

return nullptr;

}

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

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

{

Node\* local\_node = new Node(0);

Node\* local\_returned\_node = nullptr;

cout << "Initial State:\n";

printMatixNode(root);

unsetDataNodes.insert(getString(root));

priority\_queue.push(root);

//list\_allPath.push\_front(root);

local\_node = root;

printDataPath(priority\_queue);

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

fflush(stdin);

getchar();

/\*----------------------------------------------RENDERING------------------------------------------------\*/

RenderWindow window(VideoMode(300, 300), "My window");

this->RenderLoad(local\_node);

/\*----------------------------------------------RENDERING------------------------------------------------\*/

while (!priority\_queue.empty()) {

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

local\_node = priority\_queue.top();

priority\_queue.pop();

//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, true, choose);

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->down), 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->up), 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(priority\_queue);

if(!priority\_queue.empty())

local\_node = priority\_queue.top();

cout << "Prioritet: " << local\_node->priority << endl;

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

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

/\*----------------------------------------------RENDERING------------------------------------------------\*/

if (local\_node != nullptr) {

Event event;

while (window.pollEvent(event))

{

if (event.type == Event::Closed)

window.close();

}

this->RenderUpdate(local\_node);

window.clear(Color::Yellow);

this->RenderDraw(window);

window.display();

}

/\*----------------------------------------------RENDERING------------------------------------------------\*/

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->up != nullptr) {

printMatixNode(local\_node->up);

}

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

printMatixNode(local\_node->down);

}

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

printMatixNode(local\_node->right);

}

}

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

void Graph::printDataPath(Queue\_p queue\_prioritet) {

Node\* cur = new Node(0);

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

while (!queue\_prioritet.empty())

{

cur = queue\_prioritet.top();

printMatixNode(cur);

queue\_prioritet.pop();

}

}

// ---------- < 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, bool local\_steps\_output, int choose)

{

if(local\_steps\_output)cout << "\nMOVEMENT VARIANTS: ";

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) {

if(local\_steps\_output)cout << "<- ";

(\*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) {

if(local\_steps\_output)cout << "V ";

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

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

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

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

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

}

if (y > 0) {

if(local\_steps\_output)cout << "^ ";

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

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

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

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

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

}

if (y < 2) {

if(local\_steps\_output)cout << "-> ";

(\*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;

}

choiceOfHeuristic(local\_node, choose);

if(local\_steps\_output)cout << "\n";

}

// ---------- < 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->priority\_queue.push(\*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;

}

//---------- < SELECT HEURISTIC LOGIC > ----------

void Graph::choiceOfHeuristic(Node\* local\_node, int choose) {

if (choose == 1) {

f\_h1(local\_node);

}

if (choose == 2) {

f\_h2(local\_node);

}

if (choose == 3) {

f\_Ah1(local\_node);

}

if (choose == 4) {

f\_Ah2(local\_node);

}

}

//---------- <COMPARE PRIORITY OF NODE FOR H1 > ----------

int Graph::compareH1(Node\* local\_node) {

int count = 0;

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] && local\_node->elements[i][j] != ' ') {

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

count++;

}

}

}

}

}

}

return count;

}

//---------- <COMPARE PRIORITY OF NODE FOR H1 > ----------

int Graph::compareH2(Node\* local\_node) {

int count = 0;

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] && local\_node->elements[i][j] != ' ') {

count = count + abs(i - y) + abs(j - x);

}

}

}

}

}

return count;

}

//---------- < h1 LOGIC > ----------

void Graph::f\_h1(Node\* local\_node) {

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

local\_node->left->priority = compareH1(local\_node->left);

}

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

local\_node->down->priority = compareH1(local\_node->down);

}

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

local\_node->up->priority = compareH1(local\_node->up);

}

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

local\_node->right->priority = compareH1(local\_node->right);

}

}

//---------- < h2 LOGIC > ----------

void Graph::f\_h2(Node\* local\_node) {

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

local\_node->left->priority = compareH2(local\_node->left);

}

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

local\_node->down->priority = compareH2(local\_node->down);

}

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

local\_node->up->priority = compareH2(local\_node->up);

}

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

local\_node->right->priority = compareH2(local\_node->right);

}

}

//---------- <COMPARE PRIORITY OF NODE FOR A\* H1 > ----------

int Graph::compareAH1(Node\* cur) {

int count = 0;

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 (cur->elements[i][j] == end->elements[y][x] && cur->elements[i][j] != ' ') {

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

count++;

}

}

}

}

}

}

return (count + cur->depth);

}

//---------- <COMPARE PRIORITY OF NODE FOR A\* H2 > ----------

int Graph::compareAH2(Node\* cur) {

int count = 0;

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 (cur->elements[i][j] == end->elements[y][x] && cur->elements[i][j] != ' ') {

count = count + abs(i - y) + abs(j - x);

}

}

}

}

}

return (count + cur->depth);

}

//---------- <A\* h1 LOGIC > ----------

void Graph::f\_Ah1(Node\* local\_node) {

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

local\_node->left->priority = compareAH1(local\_node->left);

}

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

local\_node->down->priority = compareAH1(local\_node->down);

}

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

local\_node->up->priority = compareAH1(local\_node->up);

}

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

local\_node->right->priority = compareAH1(local\_node->right);

}

}

//---------- <A\* h2 LOGIC > ----------

void Graph::f\_Ah2(Node\* local\_node) {

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

local\_node->left->priority = compareAH2(local\_node->left);

}

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

local\_node->down->priority = compareAH2(local\_node->down);

}

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

local\_node->up->priority = compareAH2(local\_node->up);

}

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

local\_node->right->priority = compareAH2(local\_node->right);

}

}

void Graph::RenderLoad(Node\* local\_node)

{

if (font.loadFromFile("Fonts/arial.ttf"))

{

cell\_1.setFont(font);

cell\_2.setFont(font);

cell\_3.setFont(font);

cell\_4.setFont(font);

cell\_5.setFont(font);

cell\_6.setFont(font);

cell\_7.setFont(font);

cell\_8.setFont(font);

cell\_9.setFont(font);

}

else

{

std::cout << "Failed to load font" << std::endl;

}

cell\_1s.setSize(sf::Vector2f(100.0f, 100.0f));

cell\_2s.setSize(sf::Vector2f(100.0f, 100.0f));

cell\_3s.setSize(sf::Vector2f(100.0f, 100.0f));

cell\_4s.setSize(sf::Vector2f(100.0f, 100.0f));

cell\_5s.setSize(sf::Vector2f(100.0f, 100.0f));

cell\_6s.setSize(sf::Vector2f(100.0f, 100.0f));

cell\_7s.setSize(sf::Vector2f(100.0f, 100.0f));

cell\_8s.setSize(sf::Vector2f(100.0f, 100.0f));

cell\_9s.setSize(sf::Vector2f(100.0f, 100.0f));

cell\_1.setCharacterSize(75);

cell\_2.setCharacterSize(75);

cell\_3.setCharacterSize(75);

cell\_4.setCharacterSize(75);

cell\_5.setCharacterSize(75);

cell\_6.setCharacterSize(75);

cell\_7.setCharacterSize(75);

cell\_8.setCharacterSize(75);

cell\_9.setCharacterSize(75);

cell\_1.setPosition(0, 0);

cell\_2.setPosition(100, 0);

cell\_3.setPosition(200, 0);

cell\_1s.setPosition(0, 0);

cell\_2s.setPosition(100, 0);

cell\_3s.setPosition(200, 0);

cell\_4.setPosition(0, 100);

cell\_5.setPosition(100, 100);

cell\_6.setPosition(200, 100);

cell\_4s.setPosition(0, 100);

cell\_5s.setPosition(100, 100);

cell\_6s.setPosition(200, 100);

cell\_7.setPosition(0, 200);

cell\_8.setPosition(100, 200);

cell\_9.setPosition(200, 200);

cell\_7s.setPosition(0, 200);

cell\_8s.setPosition(100, 200);

cell\_9s.setPosition(200, 200);

line1.setSize(sf::Vector2f(300.0f, 5.0f));

line1.setPosition(0, 100);

line2.setSize(sf::Vector2f(300.0f, 5.0f));

line2.setPosition(0, 200);

line3.setSize(sf::Vector2f(300.0f, 5.0f));

line3.rotate(90.0f);

line3.setPosition(100, 0);

line4.setSize(sf::Vector2f(300.0f, 5.0f));

line4.rotate(90.0f);

line4.setPosition(200, 0);

}

void Graph::RenderUpdate(Node\* local\_node)

{

cell\_1.setString(local\_node->elements[0][0]);

cell\_2.setString(local\_node->elements[0][1]);

cell\_3.setString(local\_node->elements[0][2]);

cell\_4.setString(local\_node->elements[1][0]);

cell\_5.setString(local\_node->elements[1][1]);

cell\_6.setString(local\_node->elements[1][2]);

cell\_7.setString(local\_node->elements[2][0]);

cell\_8.setString(local\_node->elements[2][1]);

cell\_9.setString(local\_node->elements[2][2]);

cell\_1s.setFillColor(sf::Color::Black);

cell\_2s.setFillColor(sf::Color::Black);

cell\_3s.setFillColor(sf::Color::Black);

cell\_4s.setFillColor(sf::Color::Black);

cell\_5s.setFillColor(sf::Color::Black);

cell\_6s.setFillColor(sf::Color::Black);

cell\_7s.setFillColor(sf::Color::Black);

cell\_8s.setFillColor(sf::Color::Black);

cell\_9s.setFillColor(sf::Color::Black);

if (local\_node->elements[0][0] == ' ')

cell\_1s.setFillColor(sf::Color::Red);

if (local\_node->elements[0][1] == ' ')

cell\_2s.setFillColor(sf::Color::Red);

if (local\_node->elements[0][2] == ' ')

cell\_3s.setFillColor(sf::Color::Red);

if (local\_node->elements[1][0] == ' ')

cell\_4s.setFillColor(sf::Color::Red);

if (local\_node->elements[1][1] == ' ')

cell\_5s.setFillColor(sf::Color::Red);

if (local\_node->elements[1][2] == ' ')

cell\_6s.setFillColor(sf::Color::Red);

if (local\_node->elements[2][0] == ' ')

cell\_7s.setFillColor(sf::Color::Red);

if (local\_node->elements[2][1] == ' ')

cell\_8s.setFillColor(sf::Color::Red);

if (local\_node->elements[2][2] == ' ')

cell\_9s.setFillColor(sf::Color::Red);

}

void Graph::RenderDraw(sf::RenderWindow& window)

{

window.draw(cell\_1s);

window.draw(cell\_2s);

window.draw(cell\_3s);

window.draw(cell\_4s);

window.draw(cell\_5s);

window.draw(cell\_6s);

window.draw(cell\_7s);

window.draw(cell\_8s);

window.draw(cell\_9s);

window.draw(cell\_1);

window.draw(cell\_2);

window.draw(cell\_3);

window.draw(cell\_4);

window.draw(cell\_5);

window.draw(cell\_6);

window.draw(cell\_7);

window.draw(cell\_8);

window.draw(cell\_9);

window.draw(line1);

window.draw(line2);

window.draw(line3);

window.draw(line4);

}

**main.cpp**

#include <iostream>

#include <stack>

#include <unordered\_set>

#include <string>

#include <list>

#include "./node.h"

#include "./graph.h"

#include <SFML\Graphics.hpp>

#define \_HAS\_ITERATOR\_DEBUGGING 2

#define \_ITERATOR\_DEBUG\_LEVEL 2

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 algorithm\_settingsOption = 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;

cout << "Please, choose algoritm settings:\n"

<< "0 -- Back\n"

<< "1 -- h1\n" //кол-во фишек не на своих местах

<< "2 -- h2\n" //манхэттенское расстояние

<< "3 -- A\*(h1)\n"

<< "4 -- A\*(h2)\n"

<< "Other -- Default -- h1\n";

cin >> algorithm\_settingsOption;

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

{

graph->root->priority = 8;

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

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

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

break;

}

case 2: // idDFS

{

graph->root->priority = 14;

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, algorithm\_settingsOption);

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

{

graph->root->priority = 8;

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

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

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

break;

}

case 4: // idDFS steps

{

graph->root->priority = 14;

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, algorithm\_settingsOption);

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

{

graph->root->priority = 8;

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

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.