

**Міністерство освіти і науки України**  
**Національний технічний університет України «Київський політехнічний**  
**інститут імені Ігоря Сікорського»**  
**Факультет інформатики та обчислювальної техніки**  
  
**Кафедра інформатики та програмної інженерії**

**Звіт**

з лабораторної роботи № 2 з дисципліни  
«Проектування алгоритмів»

**«Неінформативний, інформативний та локальний пошук»**

**Виконав(ла)**

ІІ-15 Волинець Кирило Михайлович  
(шифр, прізвище, ім'я, по батькові)

**Перевірив**

Ахаладзе І.Е.  
(прізвище, ім'я, по батькові)

Київ 2022

## ЗМІСТ

<b>1</b>	<b>МЕТА ЛАБОРАТОРНОЇ РОБОТИ .....</b>	<b>3</b>
<b>2</b>	<b>ЗАВДАННЯ .....</b>	<b>4</b>
<b>3</b>	<b>ВИКОНАННЯ.....</b>	<b>6</b>
3.1	ПСЕВДОКОД АЛГОРИТМІВ.....	6
3.2	ПРОГРАМНА РЕАЛІЗАЦІЯ.....	8
3.2.1	<i>Вихідний код.....</i>	8
3.2.2	<i>Приклади роботи .....</i>	8
3.3	ДОСЛІДЖЕННЯ АЛГОРИТМІВ .....	24
	<b>ВИСНОВОК .....</b>	<b>27</b>
	<b>КРИТЕРІЇ ОЦІНЮВАННЯ .....</b>	<b>28</b>

## 1 МЕТА ЛАБОРАТОРНОЇ РОБОТИ

Мета роботи – розглянути та дослідити алгоритми неінформативного, інформативного та локального пошуку. Провести порівняльний аналіз ефективності використання алгоритмів.

## 2 ЗАВДАННЯ

Записати алгоритм розв’язання задачі у вигляді псевдокоду, відповідно до варіанту (таблиця 2.1).

Реалізувати програму, яка розв’язує поставлену задачу згідно варіанту (таблиця 2.1) за допомогою алгоритму неінформативного пошуку **АНП**, алгоритму інформативного пошуку **АІП**, що використовує задану евристичну функцію **Func**, або алгоритму локального пошуку **АЛП** та **бектрекінгу**, що використовує задану евристичну функцію **Func.D**

Програму реалізувати на довільній мові програмування.

**Увага!** Алгоритм неінформативного пошуку **АНП**, реалізовується за принципом «AS IS», тобто так, як є, без додаткових модифікацій (таких як перевірка циклів, наприклад).

Провести серію експериментів для вивчення ефективності роботи алгоритмів. Кожний експеримент повинен відрізнятися початковим станом. Серія повинна містити не менше 20 експериментів для кожного алгоритму. Початковий стан зафіксувати у таблиці експериментів. За проведеними серіями необхідно визначити:

- середню кількість етапів (кроків), які знадобилось для досягнення розв’язку (ітерації);
- середню кількість випадків, коли алгоритм потрапляв в глухий кут (не міг знайти оптимальний розв’язок) – якщо таке можливе;
- середню кількість згенерованих станів під час пошуку;
- середню кількість станів, що зберігаються в пам’яті під час роботи програми.

Передбачити можливість обмеження виконання програми за часом (30 хвилин) та використання пам’яті (1 Гб).

### Використані позначення:

– **Лабіринт** – задача пошуку шляху у довільному лабіринті від початкової точки до кінцевої з можливими випадками відсутності шляху.

Структура лабіринту зчитується з файлу, або генерується програмою.

– **IDS** – Пошук вглиб з ітеративним заглибленням.

– **A\*** – Пошук A\*.

– **H2** – Манхетенська відстань.

Таблиця 2.1 – Варіанти алгоритмів

№	Задача	АНП	АП	АЛП	Func
5	Лабіринт	IDS	A*		H2

## 3.1 Псевдокод алгоритмів

IDS(start, end)

depth = 0

**while** (true) **do**

result = DFS(start, end, depth)

**if** (result >= 0) **break**

depth += 1

**end while**

DFS(start, end, depth)

**if** (start == end) **return** depth

**else if** (depth <= 0) **return** -1

**for each** neighbor **in** start.neighbors **do**

result = DFS(neighbor, end, depth - 1)

**if** (result >= 0) **return** result + 1

**end for**

**return** -1

Astar(start, end)

priorityQueue.push\_back(start)

findAnyWay(priorityQueue, visitedTable, 0, end)

priorityQueue.erase()

result = false

**while** (priorityQueue.size() > 0) **do**

findIndex = min\_element(priorityQueue)

result = findAnyWay(priorityQueue, visitedTable, findIndex,

end);

**if** (result) **break**

```

        priorityQueue.erase(findedIndex);
    end while
    if (!result) return -1;
    bestDistance = priorityQueue[findedIndex].traveledDistance
    while (priorityQueue.size() > 0) do
        findedIndex = min_element(priorityQueue)
        result = findMinWay(priorityQueue, visitedTable, findedIndex,
end, bestDistance);
        if (result and priorityQueue[findedIndex].traveledDistance <
bestDistance) do
            bestDistance = priorityQueue[findedIndex].traveledDistance
        end if
        priorityQueue.erase(findedIndex)
    end while
    return bestDistance

```

```

findAnyWay(queue, table, index, end)
    element = queue[index];
    if (finded == end) return true
    visited[finded.y][finded.x] = 1
    neighbors = finded.getNeighbors
    for neighbor in neighbors do
        if (visited[neighbor.y][neighbor.x]) continue
        queue.push_back(neighbor)
    end for
    return false

```

```

findMinWay(queue, table, index, end, bestDistance)
    element = queue[index];
    if (finded == end) return true

```

```

visited[finded.y][finded.x] = 1
if (finded >= bestDistance) return false
neighbors = finded.getNeighbors
for neighbor in neighbors do
    if (visited[neighbor.y][neighbor.x]) continue
    queue.push_back(neighbor)
end for
return false

```

## 3.2 Програмна реалізація

### 3.2.1 Вихідний код

Alg\_2.cpp:

```

#include <iostream>
#include "Labyrinth.h"

using namespace std;

const float WALL_CHANCE = 0.5;
const float DEADEND_CHANCE = 0;

bool menu() {
    cout << "Enter type of input (0 - file, 1 - generation)\n";
    bool choice;
    cin >> choice;

    Labyrinth labyrinth;

    if (choice)
    {
        cout << "Enter size (X Y)\n";
        int sizeX, sizeY;
        cin >> sizeX >> sizeY;
        labyrinth.createTable(sizeX, sizeY);

        cout << "Enter seed (0 for random)\n";
        time_t seed;
        cin >> seed;
        if (seed == 0) seed = time(0);
        srand(seed);
        labyrinth.generate(WALL_CHANCE, DEADEND_CHANCE);
        labyrinth.print();

        cout << "\nExport to file(0/1)?\n";
        cin >> choice;
        if (choice)
        {
            cout << "Enter file name\n";
            std::string filename;
            cin >> filename;

```



```

        labyrinth.exportToFile_Text(filename);
    }
}
else {
    cout << "Enter file name\n";
    std::string filename;
    cin >> filename;
    labyrinth.importFromFile_Text(filename);
    labyrinth.print();
}
cout << "Enter start node \"X Y\"\n";
int startX, startY;
cin >> startX >> startY;
cout << "Enter end node \"X Y\"\n";
int endX, endY;
cin >> endX >> endY;

cout << "What algorithm to use? (-1 - IDS, 1 - Astar, 0 - both)\n";
int algorithm;
cin >> algorithm;
int iterations, visitednodes;
if (algorithm != -1)
{
    cout << "\n";
    int distance = Astar_debug(startX, startY, endX, endY, labyrinth, iterations,
visitednodes);
    cout << "A star distance - " << distance << endl;
    cout << "Iterations - " << iterations << endl;
    cout << "Deadends - " << labyrinth.countDeadends() << endl;
    cout << "Visited nodes - " << visitednodes << endl;
    cout << "Nodes - " << labyrinth.countAllNodes() << endl;
}
if (algorithm != 1)
{
    cout << "\n";
    int distance = IDS_debug(startX, startY, endX, endY, labyrinth, iterations,
visitednodes);
    cout << "IDS distance - " << distance << endl;
    cout << "Iterations - " << iterations << endl;
    cout << "Deadends - " << labyrinth.countDeadends() << endl;
    cout << "Visited nodes - " << visitednodes << endl;
    cout << "Nodes - " << labyrinth.countAllNodes() << endl;
}
cout << endl;
return 1;
}
}

int main()
{
    while (menu());
}

```

Labyrinth.h:

```

#pragma once
#include <vector>
#include <string>

struct Cell {
    Cell *top, *bot, *left, *right;

    int coordX, coordY;

    Cell();
    Cell(Cell* top, Cell* bot, Cell* left, Cell* right);
}

```

```

    bool operator==(const Cell& cmp) const;

    void set(Cell* top, Cell* bot, Cell* left, Cell* right, int coordX, int coordY);
    std::vector<Cell*> getPtrArray();

    void addWallTop();
    void addWallBot();
    void addWallLeft();
    void addWallRight();
};

class Labyrinth {
    unsigned int sizeX, sizeY;
    std::vector<std::vector<Cell*>> table;

    void createDeadend(int x, int y);

public:
    Labyrinth();
    Labyrinth(int sizeX, int sizeY = -1);

    void createTable(int sizeX, int sizeY);

    bool operator==(const Labyrinth& cmp) const;

    std::vector<Cell*> operator[](int i);

    void importFromFile_Text(std::string filename);
    void exportToFile_Text(std::string filename);

    void generate(float wallChance = 0.2, float deadendChance = 0.2);

    Cell* getCell(int x, int y);

    void print();

    int countDeadends();
    inline int countAllNodes() { return sizeX * sizeY; }

    friend int IDS(int startX, int startY, int endX, int endY, Labyrinth& labyrinth);
    friend int Astar(int startX, int startY, int endX, int endY, Labyrinth& labyrinth);

    friend int IDS_debug(int startX, int startY, int endX, int endY, Labyrinth&
labyrinth, int& iterations, int& visited);
    friend int Astar_debug(int startX, int startY, int endX, int endY, Labyrinth&
labyrinth, int& iterations, int& visited);
};

int IDS(int startX, int startY, int endX, int endY, Labyrinth& labyrinth);
int Astar(int startX, int startY, int endX, int endY, Labyrinth& labyrinth);

int IDS_debug(int startX, int startY, int endX, int endY, Labyrinth& labyrinth, int&
iterations, int& visited);
int Astar_debug(int startX, int startY, int endX, int endY, Labyrinth& labyrinth, int&
iterations, int& visited);

Labyrinth.cpp:

#include "Labyrinth.h"
#include <fstream>
#include <iostream>
#include <sstream>

bool getBit(int n, int bit_pos)
{

```

```

        return (n & (1 << bit_pos)) >> bit_pos;
    }

Cell::Cell()
{
    top = nullptr;
    bot = nullptr;
    left = nullptr;
    right = nullptr;

    coordX = -1;
    coordY = -1;
}

Cell::Cell(Cell* top, Cell* bot, Cell* left, Cell* right)
{
    this->top = top;
    this->bot = bot;
    this->left = left;
    this->right = right;

    coordX = -1;
    coordY = -1;
}

bool Cell::operator==(const Cell& cmpr) const
{
    if ((bool)top != (bool)cmpr.top) return false;
    if ((bool)bot != (bool)cmpr.bot) return false;
    if ((bool)left != (bool)cmpr.left) return false;
    if ((bool)right != (bool)cmpr.right) return false;

    return true;
}

void Cell::set(Cell* top, Cell* bot, Cell* left, Cell* right, int coordX, int coordY)
{
    this->top = top;
    this->bot = bot;
    this->left = left;
    this->right = right;

    this->coordX = coordX;
    this->coordY = coordY;
}

std::vector<Cell*> Cell::getPtrArray() {
    std::vector<Cell*> result;

    if (top) result.push_back(top);
    if (bot) result.push_back(bot);
    if (left) result.push_back(left);
    if (right) result.push_back(right);

    return result;
}

void Cell::addWallTop()
{
    if (top) {
        top->bot = nullptr;
        top = nullptr;
    }
}

```

```

void Cell::addWallBot()
{
    if (bot) {
        bot->top = nullptr;
        bot = nullptr;
    }
}

void Cell::addWallLeft()
{
    if (left) {
        left->right = nullptr;
        left = nullptr;
    }
}

void Cell::addWallRight()
{
    if (right) {
        right->left = nullptr;
        right = nullptr;
    }
}

Labyrinth::Labyrinth()
{
    sizeX = 0;
    sizeY = 0;
}

void initiateRow(std::vector<Cell>* topRow, std::vector<Cell>* currentRow,
std::vector<Cell>* botRow, int rowID)
{
    int size = currentRow->size();

    Cell* topRowPtr = nullptr;
    Cell* botRowPtr = nullptr;

    if (topRow) topRowPtr = topRow->begin()._Ptr;
    if (botRow) botRowPtr = botRow->begin()._Ptr;

    Cell* currentRowPtrPrev = currentRow->begin()._Ptr;
    auto currentRowPtr = currentRow->begin();

    Cell* currentRowPtrNext;
    if (currentRow->begin() + 1 == currentRow->end()) currentRowPtrNext = nullptr;
    else currentRowPtrNext = (currentRow->begin() + 1)._Ptr;

    currentRowPtr->set(topRowPtr, botRowPtr, nullptr, currentRowPtrNext, 0, rowID);
    ++currentRowPtr;
    ++currentRowPtrNext;
    if (topRowPtr) ++topRowPtr;
    if (botRowPtr) ++botRowPtr;

    for (size_t i = 2; i < size; ++i)
    {
        currentRowPtr->set(topRowPtr, botRowPtr, currentRowPtrPrev, currentRowPtrNext,
i - 1, rowID);

        ++currentRowPtrPrev;
        ++currentRowPtr;
        ++currentRowPtrNext;
        if (topRowPtr) ++topRowPtr;
        if (botRowPtr) ++botRowPtr;
    }
}

```

```

    }

    if(size > 1) currentRowPtr->set(topRowPtr, botRowPtr, currentRowPtrPrev, nullptr,
size - 1, rowID);
}

void Labyrinth::createTable(int sizeX, int sizeY)
{
    table = std::vector<std::vector<Cell>>(sizeY, std::vector<Cell>(sizeX));

    std::vector<Cell>* tablePtrPrev = table.begin()._Ptr;
    std::vector<Cell>* tablePtrCurrent = table.begin()._Ptr;
    std::vector<Cell>* tablePtrNext;
    if (table.begin() + 1 == table.end()) tablePtrNext = nullptr;
    else tablePtrNext = (table.begin() + 1)._Ptr;

    initiateRow(nullptr, tablePtrCurrent, tablePtrNext, 0);
    ++tablePtrCurrent;
    ++tablePtrNext;

    for (size_t i = 2; i < sizeY; ++i)
    {
        initiateRow(tablePtrPrev, tablePtrCurrent, tablePtrNext, i - 1);
        ++tablePtrPrev;
        ++tablePtrCurrent;
        ++tablePtrNext;
    }
    if(table.size() > 1) initiateRow(tablePtrPrev, tablePtrCurrent, nullptr, sizeY - 1);

    this->sizeX = sizeX;
    this->sizeY = sizeY;
}

Labyrinth::Labyrinth(int sizeX, int sizeY)
{
    if (sizeY < 0) sizeY = sizeX;

    createTable(sizeX, sizeY);
}

bool Labyrinth::operator==(const Labyrinth& cmpr) const
{
    if (sizeX != cmpr.sizeX or sizeY != cmpr.sizeY) return false;

    for (size_t i = 0; i < sizeY; ++i)
    {
        for (size_t j = 0; j < sizeX; ++j)
        {
            if (table[i][j] != cmpr.table[i][j]) return false;
        }
    }

    return true;
}

std::vector<Cell>& Labyrinth::operator[](int i)
{
    return table[i];
}

void Labyrinth::exportToFile_Text(std::string filename)
{
    std::ofstream file(filename);

    file << sizeX << " " << sizeY << std::endl;
}

```

```

auto rowPtr = table.begin();
for (int i = 0; i < sizeY; ++i)
{
    auto cellPtr = rowPtr->begin();
    for (int j = 1; j < sizeX; ++j)
    {
        file << !(bool)cellPtr->right << " ";
        ++cellPtr;
    }
    file << std::endl;
    ++rowPtr;
}

rowPtr = table.begin();
for (int i = 1; i < sizeY; ++i)
{
    auto cellPtr = rowPtr->begin();
    for (int j = 0; j < sizeX; ++j)
    {
        file << !(bool)cellPtr->bot << " ";
        ++cellPtr;
    }
    file << std::endl;
    ++rowPtr;
}

file.close();
}

```

```

void Labyrinth::importFromFile_Text(std::string filename)
{
    std::vector<std::vector<int>> verticalWalls;
    std::vector<std::vector<int>> horizontalWalls;

    std::ifstream file(filename);

    file >> sizeX >> sizeY;
    createTable(sizeX, sizeY);

    bool isWall = false;

    for (size_t i = 0; i < sizeY; ++i)
    {
        for (size_t j = 1; j < sizeX; ++j)
        {
            file >> isWall;
            if (isWall) {
                table[i][j].left->right = nullptr;
                table[i][j].left = nullptr;
            }
        }
    }

    for (size_t i = 1; i < sizeY; ++i)
    {
        for (size_t j = 0; j < sizeX; ++j)
        {
            file >> isWall;
            if (isWall) {
                table[i][j].top->bot = nullptr;
                table[i][j].top = nullptr;
            }
        }
    }
}

```

```

    }
}

Cell* Labyrinth::getCell(int x, int y)
{
    return &table[y][x];
}

void Labyrinth::print()
{
    for (size_t i = 0; i < sizeX; ++i)
    {
        std::cout << "  ____";
    }
    std::cout << "\n";

    auto rowPtr = table.begin();
    for (int i = 0; i < sizeY; ++i)
    {
        std::cout << "|";
        auto cellPtr = rowPtr->begin();
        for (int j = 0; j < sizeX; ++j)
        {
            if (cellPtr->bot) std::cout << "  ";
            else std::cout << "  ____";

            if (cellPtr->right) std::cout << " ";
            else std::cout << "|";

            ++cellPtr;
        }
        std::cout << "\n";
        ++rowPtr;
    }
}

int Labyrinth::countDeadends()
{
    int counter = 0;
    for (size_t i = 0; i < table.size(); ++i)
    {
        for (size_t j = 0; j < table[i].size(); ++j)
        {
            if (table[i][j].getPtrArray().size() <= 1) ++counter;
        }
    }
    return counter;
}

Generation.cpp:

#include "Labyrinth.h"

float randFloat()
{
    return (float)rand() / RAND_MAX;
}

void Labyrinth::createDeadend(int x, int y)
{
    Cell& cell = table[y][x];
    cell.addWallBot();
    cell.addWallTop();
    cell.addWallLeft();
}

```

```

        cell.addWallRight();
    }

void addVerticalWallsFirstLine(std::vector<Cell>::iterator cellPtr,
std::vector<int>::iterator MTptr, int sizeX, float wallChance, int& MTcount)
{
    for (size_t i = 0; i < sizeX - 1; ++i)
    {
        *MTptr = MTcount;
        if (randFloat() <= wallChance) {
            cellPtr->right->left = nullptr;
            cellPtr->right = nullptr;
            ++MTcount;
        }
        ++cellPtr;
        ++MTptr;
    }
    *MTptr = MTcount;
}

void addHorizontalWalls(std::vector<Cell>::iterator cellPtr, std::vector<int>::iterator
MTptr, int sizeX, float wallChance)
{
    bool haveExit = false;
    int currentMT = -1;
    for (size_t i = 0; i < sizeX - 1; ++i)
    {
        if (*MTptr != currentMT) {
            haveExit = false;
            currentMT = *MTptr;
        }

        if (cellPtr->right or haveExit) {
            if (randFloat() <= wallChance and cellPtr->bot) {
                cellPtr->bot->top = nullptr;
                cellPtr->bot = nullptr;
            }
        }
        else {
            haveExit = true;
        }

        ++cellPtr;
        ++MTptr;
    }
}

void resetMT(std::vector<Cell>::iterator cellPtr, std::vector<int>::iterator MTptr, int
sizeX, float wallChance, int& MTcount)
{
    for (size_t i = 0; i < sizeX; ++i)
    {
        if (!(cellPtr->top)) {
            *MTptr = MTcount;
            ++MTcount;
        }
        ++cellPtr;
        ++MTptr;
    }
}

void addVerticalWalls(std::vector<Cell>::iterator cellPtr, std::vector<int>::iterator
MTptrNext, int sizeX, float wallChance, int& MTcount)
{

```



```

    auto MTptrCurrent = MTptrNext;
    ++MTptrNext;
    for (size_t i = 0; i < sizeX - 1; ++i)
    {
        if (*MTptrNext == *MTptrCurrent) {
            cellPtr->right->left = nullptr;
            cellPtr->right = nullptr;
        }
        else if (randFloat() <= wallChance) {
            cellPtr->right->left = nullptr;
            cellPtr->right = nullptr;
        }
        else {
            *MTptrNext = *MTptrCurrent;
        }
        ++cellPtr;
        ++MTptrNext;
        ++MTptrCurrent;
    }
}

void addVerticalWallsLastLine(std::vector<Cell>::iterator cellPtr,
std::vector<int>::iterator MTptrNext, int sizeX, float wallChance, int& MTcount)
{
    auto MTptrCurrent = MTptrNext;
    ++MTptrNext;
    for (size_t i = 0; i < sizeX - 1; ++i)
    {
        if (*MTptrNext == *MTptrCurrent and randFloat() <= wallChance) {
            cellPtr->right->left = nullptr;
            cellPtr->right = nullptr;
        }
        ++cellPtr;
        ++MTptrNext;
        ++MTptrCurrent;
    }
}

void generateFirstLine(std::vector<Cell>& firstLine, std::vector<int>& MTline, int& MTcount,
float wallChance)
{
    addVerticalWallsFirstLine(firstLine.begin(), MTline.begin(), firstLine.size(),
wallChance, MTcount);
    addHorizontalWalls(firstLine.begin(), MTline.begin(), firstLine.size(), wallChance);
}

void generateLine(std::vector<Cell>& tableLine, std::vector<int>& MTline, int& MTcount,
float wallChance)
{
    resetMT(tableLine.begin(), MTline.begin(), tableLine.size(), wallChance, MTcount);
    addVerticalWalls(tableLine.begin(), MTline.begin(), tableLine.size(), wallChance,
MTcount);
    addHorizontalWalls(tableLine.begin(), MTline.begin(), tableLine.size(), wallChance);
}

void generateLastLine(std::vector<Cell>& lastLine, std::vector<int>& MTline, int& MTcount,
float wallChance)
{
    resetMT(lastLine.begin(), MTline.begin(), lastLine.size(), wallChance, MTcount);
    addVerticalWallsLastLine(lastLine.begin(), MTline.begin(), lastLine.size(),
wallChance, MTcount);
}

std::vector<std::vector<Cell>>::iterator;

```

```

void Labyrinth::generate(float wallChance, float deadendChance)
{
    int MT_count = 0;
    std::vector<std::vector<int>> MT(sizeY, std::vector<int>(sizeX, -1));

    if(sizeY > 1) generateFirstLine(table[0], MT[0], MT_count, wallChance);

    for (size_t i = 1; i < sizeY - 1; ++i)
    {
        std::copy(MT[i - 1].begin(), MT[i - 1].end(), MT[i].begin());
        generateLine(table[i], MT[i], MT_count, wallChance);
    }

    if(sizeY > 1) std::copy(MT[sizeY - 2].begin(), MT[sizeY - 2].end(), MT[sizeY - 1].begin());
    generateLastLine(table[sizeY - 1], MT[sizeY - 1], MT_count, wallChance);

    if (randFloat() <= deadendChance) createDeadend(sizeX - 1, sizeY - 1);
}

```

IDS.cpp:

```

#include "Labyrinth.h"
#include <iostream>

int DFS(Cell* start, Cell* end, int depth)
{
    if (start == end) return depth;
    else if (depth <= 0) return -1;

    auto targetArray = start->getPtrArray();
    int result;

    for (auto targetArrayIterator = targetArray.begin(); targetArrayIterator != targetArray.end(); ++targetArrayIterator) {
        result = DFS(*targetArrayIterator, end, depth - 1);
        if (result >= 0) return result + 1;
    }

    return -1;
}

int IDS(int startX, int startY, int endX, int endY, Labyrinth& labyrinth)
{
    Cell* startCell = labyrinth.getCell(startX, startY);
    Cell* endCell = labyrinth.getCell(endX, endY);

    int depth = 0, result;
    while (true)
    {
        result = DFS(startCell, endCell, depth);
        std::cout << "Depth " << depth << " done!\n";
        if (result >= 0) break;
        ++depth;
    }
    return result;
}

```

Astar.cpp:

```

#include "Labyrinth.h"
#include <iostream>
#include <vector>

```

```

#include <algorithm>

struct Variant
{
    Cell* ptr;
    int traveledDistance, euristicDistance;
    Variant(Cell* _ptr, int _traveledDistance, int _euristicDistance) :ptr(_ptr),
traveledDistance(_traveledDistance), euristicDistance(_euristicDistance)
    {}
    inline int x() { return ptr->coordX; }
    inline int y() { return ptr->coordY; }
};

inline int calculateEuristic(Cell* cellPtr, int endX, int endY) { return abs(endX + endY -
cellPtr->coordX - cellPtr->coordY); }
inline int calculateEuristic(int startX, int startY, int endX, int endY) { return abs(endX +
endY - startX - startY); }

typedef __int8 Bool;
typedef std::vector<std::vector<Bool>> binary_table;

bool findAnyWay(std::vector<Variant>& queue, binary_table& visited, int index, int endX, int
endY)
{
    auto finded = queue[index];

    if (finded.x() == endX and finded.y() == endY) return true;

    visited[finded.y()][finded.x()] = 1;

    auto adjoining = finded.ptr->getPtrArray();
    auto adjoiningIter = adjoining.begin();

    for (size_t i = 0; i < adjoining.size(); ++i)
    {
        if (visited[(*adjoiningIter)->coordY][( *adjoiningIter)->coordX]) {
++adjoiningIter; continue; }

        queue.push_back(Variant(*(adjoiningIter._Ptr), finded.traveledDistance + 1,
calculateEuristic(*adjoiningIter, endX, endY)));
        ++adjoiningIter;
    }
    return false;
}

bool findMinWay(std::vector<Variant>& queue, binary_table& visited, int index, int endX, int
endY, int bestDistance)
{
    auto finded = queue[index];

    if (finded.x() == endX and finded.y() == endY) return true;

    visited[finded.y()][finded.x()] = 1;

    if (finded.euristicDistance + finded.traveledDistance >= bestDistance) return false;

    auto adjoining = finded.ptr->getPtrArray();
    auto adjoiningIter = adjoining.begin();

    for (size_t i = 0; i < adjoining.size(); ++i)
    {
        if (visited[( *adjoiningIter)->coordY][( *adjoiningIter)->coordX]) continue;

        queue.push_back(Variant(*(adjoiningIter._Ptr), finded.traveledDistance + 1,
calculateEuristic(*adjoiningIter, endX, endY)));
    }
}

```

```

        ++adjoiningIter;
    }
    return false;
}

int Astar(int startX, int startY, int endX, int endY, Labyrinth& labyrinth)
{
    std::vector<Variant> priorityQueue;
    binary_table visitedTable(labyrinth.sizeY, std::vector<Bool>(labyrinth.sizeX, 0));

    priorityQueue.push_back(Variant(&labyrinth[startY][startX], 0,
calculateEuristic(startX, startY, endX, endY)));
    bool result = findAnyWay(priorityQueue, visitedTable, 0, endX, endY);
    if (result) return 0;
    priorityQueue.erase(priorityQueue.begin());

    int findedIndex;
    while (priorityQueue.size() > 0)
    {
        findedIndex = std::min_element(priorityQueue.begin(), priorityQueue.end(),
        [](const Variant& v1, const Variant& v2) -> bool {
            return v1.euristicDistance + v1.traveledDistance <
v2.euristicDistance + v2.traveledDistance; })
        - priorityQueue.begin();
        result = findAnyWay(priorityQueue, visitedTable, findedIndex, endX, endY);
        if (result) break;
        priorityQueue.erase(priorityQueue.begin() + findedIndex);
    }

    if (!result) return -1;

    int bestDistance = priorityQueue[findedIndex].traveledDistance;

    while (priorityQueue.size() > 0)
    {
        findedIndex = std::min_element(priorityQueue.begin(), priorityQueue.end(),
        [](const Variant& v1, const Variant& v2) -> bool {
            return v1.euristicDistance + v1.traveledDistance <
v2.euristicDistance + v2.traveledDistance; })
        - priorityQueue.begin();
        result = findMinWay(priorityQueue, visitedTable, findedIndex, endX, endY,
bestDistance);
        if (result and priorityQueue[findedIndex].traveledDistance < bestDistance)
bestDistance = priorityQueue[findedIndex].traveledDistance;
        priorityQueue.erase(priorityQueue.begin() + findedIndex);
    }

    return bestDistance;
}

```

### 3.2.1 Приклади роботи

На рисунках 3.1 і 3.2 показані приклади роботи програми для різних алгоритмів пошуку.

### Рисунок 3.1 – Алгоритм IDS

```
Enter type of input (0 - file, 1 - generation)
1
Enter size (X Y)
10 10
Enter seed (0 for random)
2

|_|_|_|_|_|_|_|_|_|_|
|_|_|_|_|_|_|_|_|_|_|
|_|_|_|_|_|_|_|_|_|_|
|_|_|_|_|_|_|_|_|_|_|
|_|_|_|_|_|_|_|_|_|_|
|_|_|_|_|_|_|_|_|_|_|
|_|_|_|_|_|_|_|_|_|_|
|_|_|_|_|_|_|_|_|_|_|
|_|_|_|_|_|_|_|_|_|_|
|_|_|_|_|_|_|_|_|_|_|

Export to file(0/1)?
0
Enter start node "X Y"
0 0
Enter end node "X Y"
9 9
What algorithm to use? (-1 - IDS, 1 - Astar, 0 - both)
-1

Depth 0 done!
Depth 1 done!
Depth 2 done!
Depth 3 done!
Depth 4 done!
Depth 5 done!
Depth 6 done!
Depth 7 done!
Depth 8 done!
Depth 9 done!
Depth 10 done!
Depth 11 done!
Depth 12 done!
Depth 13 done!
Depth 14 done!
Depth 15 done!
Depth 16 done!
Depth 17 done!
Depth 18 done!
Depth 19 done!
Depth 20 done!
```

```

Depth 20 done!

Path:
0-0 0-1 1-1 1-0 2-0 3-0 4-0 5-0 6-0 6-1 6-2 7-2 7-3 8-3 8-4 8-5 9-5 9-6 9-7 9-8 9-9
IDS distance - 20
Iterations - 72407399
Deadends - 13
Visited nodes - 100
Nodes - 100

```

Рисунок 3.2 – Алгоритм A\*

```

Enter type of input (0 - file, 1 - generation)
1
Enter size (X Y)
10 10
Enter seed (0 for random)
2

  | | | | | | | | | |
  | | | | | | | | | |
  | | | | | | | | | |
  | | | | | | | | | |
  | | | | | | | | | |
  | | | | | | | | | |
  | | | | | | | | | |
  | | | | | | | | | |
  | | | | | | | | | |
  | | | | | | | | | |

Export to file(0/1)?
0
Enter start node "X Y"
0 0
Enter end node "X Y"
9 9
What algorithm to use? (-1 - IDS, 1 - Astar, 0 - both)
1

A star distance - -1
Iterations - 201
Deadends - 15
Visited nodes - 99
Nodes - 100

```

### 3.3 Дослідження алгоритмів

В таблиці 3.1 наведені характеристики оцінювання алгоритму Назва алгоритму, задачі Назва задачі для 20 початкових станів.

Таблиця 3.1 – Характеристики оцінювання алгоритму IDS

Початкові стани	Шлях	Перевірок клітинки	К-сть гл. кутів	Перевіраних клітинок	Всього клітинок
Стан 1	18	1'459'612	17	89	100
Стан 2	20	72'407'399	13	100	100
Стан 3	22	222'398'247	16	86	100
Стан 4	22	46'091'550	18	55	100
Стан 5	18	2'864'665	9	81	100
Стан 6	20	18'031'500	14	97	100
Стан 7	18	6'301'362	18	89	100
Стан 8	18	1'656'172	17	84	100
Стан 9	20	40'908'758	18	100	100
Стан 10	20	15'602'458	23	99	100
Стан 11	18	284'579	14	51	100
Стан 12	24	238'263'763	19	69	100
Стан 13	18	421'977	19	80	100
Стан 14	18	3'737'327	21	96	100
Стан 15	18	21'590'340	19	81	100
Стан 16	18	3'185'759	16	75	100
Стан 17	18	8'857'466	20	67	100
Стан 18	18	2'647'360	17	85	100
Стан 19	18	5'854'270	21	82	100
Стан 20	18	399'784	16	41	100

В таблиці 3.2 наведені характеристики оцінювання алгоритму Назва алгоритму, задачі Назва задачі для 20 початкових станів.



Таблиця 3.3 – Характеристики оцінювання алгоритму А\*

Початкові стани	Шлях	Перевірок клітинки	К-сть гл. кутів	Перевіраних клітинок	Всього клітинок
Стан 1	18	33	17	30	100
Стан 2	20	133	13	73	100
Стан 3	22	98	16	64	100
Стан 4	22	52	18	39	100
Стан 5	18	41	9	31	100
Стан 6	20	78	14	67	100
Стан 7	18	62	18	52	100
Стан 8	18	50	17	33	100
Стан 9	20	179	18	61	100
Стан 10	20	93	23	55	100
Стан 11	18	37	14	23	100
Стан 12	24	64	19	43	100
Стан 13	18	50	19	42	100
Стан 14	18	85	21	63	100
Стан 15	18	162	19	42	100
Стан 16	18	68	16	48	100
Стан 17	18	80	20	47	100
Стан 18	18	57	17	43	100
Стан 19	18	91	21	49	100
Стан 20	18	26	16	24	100

Як ми бачимо, алгоритм А\* у тисячі разів ефективніший за алгоритм IDS. При чому він ефективніший у будь-якому лабіринті і за всіма параметрами. Причин цьому декілька:

1) Для лабіринту 10x10 мінімальна відстань між кутами це 18, але алгоритм IDS все одно перебирає всі варіанти шляхів від 1 до 17, що вже займає у тисячі разів більше часу ніж повний пошук шляху у A\*.

2) Алгоритм A\* перевіряє спочатку найвірогідніший варіант шляху, а якщо навіть він невірний, то вірний варіант зазвичай знаходиться не дуже далеко. Алгоритм IDS перевіряє всі шляхи, навіть ті, які не можуть бути вірними ніяким чином.

3) Алгоритм IDS ніяк не перевіряє цикли а тому більшість варіантів шляхів, які перевіряє алгоритм, - зайві.

## ВИСНОВОК

При виконанні даної лабораторної роботи було розглянуто алгоритми інформативного та неінформативного пошуків на прикладі задачі Лабіринт. Було створено відповідний клас та алгоритм його генерації і зберігання у файл. Реалізовані алгоритми IDS та A\* для знаходження найкоротшого шляху між двома точками у лабіринті. Дослідженні відмінності цих двох алгоритмів.

## КРИТЕРІЇ ОЦІНЮВАННЯ

За умови здачі лабораторної роботи до 23.10.2022 включно максимальний бал дорівнює – 5. Після 23.10.2022 максимальний бал дорівнює – 1.

Критерії оцінювання у відсотках від максимального балу:

- псевдокод алгоритму – 10%;
- програмна реалізація алгоритму – 60%;
- дослідження алгоритмів – 25%;
- висновок – 5%.