**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)**

(шифр, прізвище, ім'я, по батькові)

ІП-11 Боровков І.І.

Київ 2022

Зміст

[1 Мета лабораторної роботи 3](#_Toc81070688)

[2 ЗаВдання 4](#_Toc81070689)

[3 Виконання 8](#_Toc81070690)

[3.1 Псевдокод алгоритмів 8](#_Toc81070691)

[3.2 Програмна реалізація 8](#_Toc81070692)

[3.2.1 Вихідний код 8](#_Toc81070693)

[3.2.2 Приклади роботи 8](#_Toc81070694)

[3.3 Дослідження алгоритмів 8](#_Toc81070695)

[Висновок 11](#_Toc81070696)

[Критерії оцінювання 12](#_Toc81070697)

# Мета лабораторної роботи

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

# 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.

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 відрізнятись 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:**

* **8-ферзів** – Задача про вісім ферзів полягає в такому розміщенні восьми ферзів на шахівниці, що жодна з них не ставить під удар один одного. Тобто, вони не повинні стояти в одній вертикалі, горизонталі чи діагоналі.
* **8-puzzle** – гра, що складається з 8 однакових квадратних пластинок з нанесеними числами від 1 до 8. Пластинки поміщаються в квадратну коробку, довжина сторони якої в три рази більша довжини сторони пластинок, відповідно в коробці залишається незаповненим одне квадратне поле. Мета гри – переміщаючи пластинки по коробці досягти впорядковування їх по номерах, бажано зробивши якомога менше переміщень.
* **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.
* **LDFS** – Пошук вглиб з обмеженням глибини.
* **BFS** – Пошук вшир.
* **IDS** – Пошук вглиб з ітеративним заглибленням.
* **A\*** – Пошук А\*.
* **RBFS** – Рекурсивний пошук за першим найкращим співпадінням.
* **F1** – кількість пар ферзів, які б’ють один одного з урахуванням видимості (ферзь А може стояти на одній лінії з ферзем В, проте між ними стоїть ферзь С; тому А не б’є В).
* **F2** – кількість пар ферзів, які б’ють один одного без урахування видимості.
* **H1** – кількість фішок, які не стоять на своїх місцях.
* **H2** – Манхетенська відстань.
* **H3** – Евклідова відстань.
* **COLOR** – Задача розфарбування карти самостійно обраної країни, не менше 20 регіонів (областей). Необхідно розфарбувати карту не більше ніж у 4 різні кольори. Мається на увазі приписування кожному регіону власного кольору так, щоб кольори сусідніх регіонів відрізнялись. Використовувати евристичну функцію, яка повертає кількість пар суміжних вузлів, що мають однаковий колір (тобто кількість конфліктів). Реалізувати алгоритм пошуку із поверненнями (backtracking) для розв’язання поставленої задачі. Для підвищення швидкодії роботи алгоритму використати евристичну функцію, а початковим станом вважати випадкову вершину.
* **HILL –** Пошук зі сходженням на вершину з використанням із використанням руху вбік (на 100 кроків) та випадковим перезапуском (кількість необхідних разів запуску визначити самостійно).
* **ANNEAL** – Локальний пошук із симуляцією відпалу. Робоча характеристика – залежність температури Т від часу роботи алгоритму t. Можна розглядати лінійну залежність: T = 1000 - k∙t, де k – змінний коефіцієнт.
* **BEAM** – Локальний променевий пошук. Робоча характеристика – кількість променів k. Експерименти проводи із кількістю променів від 2 до 21.
* **MRV** – евристика мінімальної кількості значень;
* **DGR** – ступенева евристика.

Таблиця 2.1 – Варіанти алгоритмів

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **№** | **Задача** | **АНП** | **АІП** | **АЛП** | **Func** |
| 1 | Лабіринт | LDFS | A\* |  | H2 |
| 2 | Лабіринт | LDFS | RBFS |  | H3 |
| 3 | Лабіринт | BFS | A\* |  | H2 |
| 4 | Лабіринт | BFS | RBFS |  | H3 |
| 5 | Лабіринт | IDS | A\* |  | H2 |
| 6 | Лабіринт | IDS | RBFS |  | H3 |
| 7 | 8-ферзів | LDFS | A\* |  | F1 |
| 8 | 8-ферзів | LDFS | A\* |  | F2 |
| 9 | 8-ферзів | LDFS | RBFS |  | F1 |
| 10 | 8-ферзів | LDFS | RBFS |  | F2 |
| 11 | 8-ферзів | BFS | A\* |  | F1 |
| 12 | 8-ферзів | BFS | A\* |  | F2 |
| 13 | 8-ферзів | BFS | RBFS |  | F1 |
| 14 | 8-ферзів | BFS | RBFS |  | F2 |
| 15 | 8-ферзів | IDS | A\* |  | F1 |
| 16 | 8-ферзів | IDS | A\* |  | F2 |
| 17 | 8-ферзів | IDS | RBFS |  | F1 |
| 18 | Лабіринт | LDFS | A\* |  | H3 |
| 19 | 8-puzzle | LDFS | A\* |  | H1 |
| 20 | 8-puzzle | LDFS | A\* |  | H2 |
| 21 | 8-puzzle | LDFS | RBFS |  | H1 |
| 22 | 8-puzzle | LDFS | RBFS |  | H2 |
| 23 | 8-puzzle | BFS | A\* |  | H1 |
| 24 | 8-puzzle | BFS | A\* |  | H2 |
| 25 | 8-puzzle | BFS | RBFS |  | H1 |
| 26 | 8-puzzle | BFS | RBFS |  | H2 |
| 27 | Лабіринт | BFS | A\* |  | H3 |
| 28 | 8-puzzle | IDS | A\* |  | H2 |
| 29 | 8-puzzle | IDS | RBFS |  | H1 |
| 30 | 8-puzzle | IDS | RBFS |  | H2 |
| 31 | COLOR |  |  | HILL | MRV |
| 32 | COLOR |  |  | ANNEAL | MRV |
| 33 | COLOR |  |  | BEAM | MRV |
| 34 | COLOR |  |  | HILL | DGR |
| 35 | COLOR |  |  | ANNEAL | DGR |
| 36 | COLOR |  |  | BEAM | DGR |

# Виконання

## Псевдокод алгоритмів

**LDFS(**StartNode**,** LookedNode, limit**):**

NodeStack**.push(**StartNode);

**while(not (**NodeStack.**empty() or** LookedNode == **Visited) and not** limitOversize**)**

**do**

currNode = NodeStack.**top();**

NodeStack.**pop();**

**if(isPassTop(**currNode**)**):

NodeStack.**push(**TopNode**);**

**end;**

**if(isPassLeft(**currNode**)**):

NodeStack.**push(**LeftNode**);**

**end;**

**if(isPassBottom(**currNode**)**):

NodeStack.**push(**BottomNode**);**

**end;**

**if(isPassRight(**currNode**)**):

NodeStack.**push(**RightNode**);**

**end;**

**end repeat;**

**end.**

**RBFS(**StartNode**,** LookedNode**):**

SavedNode**.push(**StartNode);

**while(not (**NodeStack.**empty() or** LookedNode == **Visited))**

**do**

currNode = SavedNode.**minValue();**

SavedNode.**delete(**SavedNode.**minValue());**

**if(isPassTop(**currNode**)**):

SavedNode.**push(**TopNode**);**

**end;**

**if(isPassLeft(**currNode**)**):

SavedNode.**push(**LeftNode**);**

**end;**

**if(isPassBottom(**currNode**)**):

SavedNode.**push(**BottomNode**);**

**end;**

**if(isPassRight(**currNode**)**):

SavedNode.**push(**RightNode**);**

**end;**

**end repeat;**

**end.**

## Програмна реалізація

### Вихідний код

**Labyirinth.h**

#pragma once  
#include <vector>  
  
struct FInfo  
{  
 size\_t Iteration = 0, GeneratedNodes = 0, SavedNodes = 0;  
};  
  
class FLabyrinth  
{  
private:  
 std::vector<std::vector<int>> Field;  
 const size\_t SizeX, SizeY;  
 FInfo Info;  
 bool IsDeadEnd(size\_t X, size\_t Y) const;  
 bool IsLeftEnd(size\_t X, size\_t Y) const;  
 bool IsRightEnd(size\_t X, size\_t Y) const;  
 bool IsTopEnd(size\_t X, size\_t Y) const;  
 bool IsBottomEnd(size\_t X, size\_t Y) const;  
 void CreatePass(size\_t& X, size\_t& Y, size\_t Course);  
  
 struct FNodeLDFS  
 {  
 size\_t X, Y, Depth;  
 };  
  
 struct FNodeRBFS  
 {  
 size\_t X, Y, Depth;  
 double Value;  
 };  
  
 bool IsPass(size\_t X, size\_t Y) const;  
 void FindWay();  
 static bool Compare(const FNodeRBFS& A, const FNodeRBFS& B);  
public:  
 FLabyrinth(size\_t SizeX, size\_t SizeY);  
 void GenerateNewLabyrinth();  
 void PrintLabyrinth() const;  
 int LDFS(size\_t MaxDepth);  
 bool RBFS();  
 double CountEvklidDistance(size\_t X, size\_t Y) const;  
 FInfo GetInfo() const;  
};

**Labyrinth.cpp**

*// ReSharper disable CppClangTidyConcurrencyMtUnsafe*  
#include "Labyrinth.h"  
#include <algorithm>  
#include <iomanip>  
#include <stack>  
  
#include <iostream>  
  
#define PASS\_SIZE 5  
  
#define WAY (-3)  
#define WALL (-1)  
#define PASS (-2)  
#define UNVISITED (-2)  
#define TOP 0  
#define RIGHT 1  
#define BOTTOM 2  
#define LEFT 3  
  
FLabyrinth::FLabyrinth(const size\_t SizeX, const size\_t SizeY) : SizeX(SizeX), SizeY(SizeY)  
{  
 GenerateNewLabyrinth();  
}  
  
void FLabyrinth::GenerateNewLabyrinth()  
{  
 Field = std::vector<std::vector<int>>(SizeX, std::vector<int>(SizeY, WALL));  
 const size\_t NumOfCrossroads = (SizeX - 1) \* (SizeY - 1) / 4;  
 size\_t NumOfVisitedCrossroads = 1;  
  
 size\_t X = 1, Y = 1;  
 while (NumOfCrossroads > NumOfVisitedCrossroads)  
 {  
 Field[X][Y] = PASS;  
 while (!IsDeadEnd(X, Y))  
 {  
 const int Course = rand() % 4;  
 CreatePass(X, Y, Course);  
 NumOfVisitedCrossroads++;  
 }  
 do  
 {  
 X = 2 \* (rand() % ((SizeX - 1) / 2)) + 1;  
 Y = 2 \* (rand() % ((SizeY - 1) / 2)) + 1;  
 }  
 while (Field[X][Y] != PASS);  
 }  
}  
  
void FLabyrinth::PrintLabyrinth() const  
{  
 for (size\_t Y = 0; Y < SizeY; ++Y)  
 {  
 for (size\_t X = 0; X < SizeX; ++X)  
 {  
 if (Field[X][Y] == PASS)  
 std::cout << ' ' << std::setw(PASS\_SIZE);  
 else if (Field[X][Y] == WALL)  
 std::cout << 'M' << std::setw(PASS\_SIZE);  
 else if (Field[X][Y] == WAY)  
 {  
 std::cout << '-' << std::setw(PASS\_SIZE);  
 }  
 else  
 std::cout << Field[X][Y] << std::setw(PASS\_SIZE);  
 }  
 std::cout << '\n';  
 }  
}  
  
bool FLabyrinth::IsDeadEnd(const size\_t X, const size\_t Y) const  
{  
 return IsLeftEnd(X, Y) && IsRightEnd(X, Y) && IsTopEnd(X, Y) && IsBottomEnd(X, Y);  
}  
  
bool FLabyrinth::IsBottomEnd(const size\_t X, const size\_t Y) const  
{  
 return (Y == SizeY - 2) || (Field[X][Y + 2] == PASS);  
}  
  
bool FLabyrinth::IsLeftEnd(const size\_t X, const size\_t Y) const  
{  
 return (X == 1) || (Field[X - 2][Y] == PASS);  
}  
  
bool FLabyrinth::IsRightEnd(const size\_t X, const size\_t Y) const  
{  
 return (X == SizeX - 2) || (Field[X + 2][Y] == PASS);  
}  
  
bool FLabyrinth::IsTopEnd(const size\_t X, const size\_t Y) const  
{  
 return (Y == 1) || (Field[X][Y - 2] == PASS);  
}  
  
void FLabyrinth::CreatePass(size\_t& X, size\_t& Y, size\_t Course)  
{  
 bool PassCreated = false;  
 while (!PassCreated)  
 {  
 switch (Course)  
 {  
 case TOP:  
 if (!IsTopEnd(X, Y))  
 {  
 Field[X][Y - 1] = PASS;  
 Field[X][Y - 2] = PASS;  
 Y -= 2;  
 PassCreated = true;  
 }  
 break;  
  
 case RIGHT:  
 if (!IsRightEnd(X, Y))  
 {  
 Field[X + 1][Y] = PASS;  
 Field[X + 2][Y] = PASS;  
 X += 2;  
 PassCreated = true;  
 }  
 break;  
  
 case BOTTOM:  
 if (!IsBottomEnd(X, Y))  
 {  
 Field[X][Y + 1] = PASS;  
 Field[X][Y + 2] = PASS;  
 Y += 2;  
 PassCreated = true;  
 }  
  
 break;  
  
 case LEFT:  
 if (!IsLeftEnd(X, Y))  
 {  
 Field[X - 1][Y] = PASS;  
 Field[X - 2][Y] = PASS;  
 X -= 2;  
 PassCreated = true;  
 }  
  
 break;  
  
 default:  
 std::cout << "Wrong Way";  
 }  
 Course = (Course + 1) % 4;  
 }  
}  
  
int FLabyrinth::LDFS(const size\_t MaxDepth)  
{  
 FNodeLDFS CurrNode{};  
 const FNodeLDFS LookedNode{SizeX - 2, SizeY - 2, 0};  
 std::stack<FNodeLDFS> NodeStack;  
 NodeStack.push(FNodeLDFS{1, 1, 1});  
  
 while (!NodeStack.empty() && Field[LookedNode.X][LookedNode.Y] == UNVISITED)  
 {  
 CurrNode = NodeStack.top();  
 const bool bIsLimitOversize = CurrNode.Depth >= MaxDepth;  
 NodeStack.pop();  
 if (IsPass(CurrNode.X - 1, CurrNode.Y) && !bIsLimitOversize)  
 {  
 NodeStack.push(FNodeLDFS{CurrNode.X - 1, CurrNode.Y, CurrNode.Depth + 1});  
 Info.GeneratedNodes++;  
 }  
 if (IsPass(CurrNode.X + 1, CurrNode.Y) && !bIsLimitOversize)  
 {  
 NodeStack.push(FNodeLDFS{CurrNode.X + 1, CurrNode.Y, CurrNode.Depth + 1});  
 Info.GeneratedNodes++;  
 }  
 if (IsPass(CurrNode.X, CurrNode.Y - 1) && !bIsLimitOversize)  
 {  
 NodeStack.push(FNodeLDFS{CurrNode.X, CurrNode.Y - 1, CurrNode.Depth + 1});  
 Info.GeneratedNodes++;  
 }  
 if (IsPass(CurrNode.X, CurrNode.Y + 1) && !bIsLimitOversize)  
 {  
 NodeStack.push(FNodeLDFS{CurrNode.X, CurrNode.Y + 1, CurrNode.Depth + 1});  
 Info.GeneratedNodes++;  
 }  
 Info.SavedNodes = std::max(Info.SavedNodes, NodeStack.size());  
 Field[CurrNode.X][CurrNode.Y] = ++Info.Iteration;  
 }  
  
 *// if (Field[LookedNode.X][LookedNode.Y] != UNVISITED)*  
 *// FindWay();*  
  
if (CurrNode.Depth >= MaxDepth)  
 return 2;  
 return Field[LookedNode.X][LookedNode.Y] == UNVISITED;  
}  
  
void FLabyrinth::FindWay()  
{  
 size\_t X = SizeX - 2, Y = SizeY - 2;  
 while (X != 1 || Y != 1)  
 {  
 if (Field[X - 1][Y] < Field[X][Y] && Field[X - 1][Y] > 0)  
 Field[X--][Y] = WAY;  
 if (Field[X + 1][Y] < Field[X][Y] && Field[X + 1][Y] > 0)  
 Field[X++][Y] = WAY;  
 if (Field[X][Y - 1] < Field[X][Y] && Field[X][Y - 1] > 0)  
 Field[X][Y--] = WAY;  
 if (Field[X][Y + 1] < Field[X][Y] && Field[X][Y + 1] > 0)  
 Field[X][Y++] = WAY;  
 }  
 Field[X][Y] = WAY;  
}  
  
bool FLabyrinth::IsPass(const size\_t X, const size\_t Y) const  
{  
 return Field[X][Y] == PASS;  
}  
  
double FLabyrinth::CountEvklidDistance(size\_t X, size\_t Y) const  
{  
 return sqrt(pow(SizeX - X, 2) + pow(SizeY - Y, 2));  
}  
  
bool FLabyrinth::Compare(const FNodeRBFS& A, const FNodeRBFS& B)  
{  
 return A.Value < B.Value;  
}  
  
bool FLabyrinth::RBFS()  
{  
 std::vector<FNodeRBFS> SavedNodes;  
 const FNodeRBFS LookedNode{SizeX - 2, SizeY - 2, 0, 0};  
 SavedNodes.push\_back(FNodeRBFS{1, 1, 1, CountEvklidDistance(1, 1)});  
 while (!SavedNodes.empty() && Field[LookedNode.X][LookedNode.Y] == UNVISITED)  
 {  
 auto CurrNodePos = std::min\_element(SavedNodes.begin(), SavedNodes.end(), Compare);  
 const FNodeRBFS CurrNode = \*CurrNodePos;  
 SavedNodes.erase(CurrNodePos);  
 if (IsPass(CurrNode.X - 1, CurrNode.Y))  
 {  
 SavedNodes.push\_back(FNodeRBFS{  
 CurrNode.X - 1, CurrNode.Y, CurrNode.Depth + 1, CountEvklidDistance(CurrNode.X - 1, CurrNode.Y)  
 });  
 ++Info.GeneratedNodes;  
 }  
 if (IsPass(CurrNode.X + 1, CurrNode.Y))  
 {  
 SavedNodes.push\_back(FNodeRBFS{  
 CurrNode.X + 1, CurrNode.Y, CurrNode.Depth + 1, CountEvklidDistance(CurrNode.X + 1, CurrNode.Y)  
 });  
 ++Info.GeneratedNodes;  
 }  
 if (IsPass(CurrNode.X, CurrNode.Y - 1))  
 {  
 SavedNodes.push\_back(FNodeRBFS{  
 CurrNode.X, CurrNode.Y - 1, CurrNode.Depth + 1, CountEvklidDistance(CurrNode.X, CurrNode.Y - 1)  
 });  
 ++Info.GeneratedNodes;  
 }  
 if (IsPass(CurrNode.X, CurrNode.Y + 1))  
 {  
 SavedNodes.push\_back(FNodeRBFS{  
 CurrNode.X, CurrNode.Y + 1, CurrNode.Depth + 1, CountEvklidDistance(CurrNode.X, CurrNode.Y + 1)  
 });  
 ++Info.GeneratedNodes;  
 }  
 Info.SavedNodes = std::max(Info.SavedNodes, SavedNodes.size());  
 Field[CurrNode.X][CurrNode.Y] = ++Info.Iteration;  
 }  
 *// if (Field[LookedNode.X][LookedNode.Y] != UNVISITED)*  
 *// FindWay();*  
return Field[LookedNode.X][LookedNode.Y] != UNVISITED;  
}  
  
FInfo FLabyrinth::GetInfo() const  
{  
 return Info;  
}

### Приклади роботи

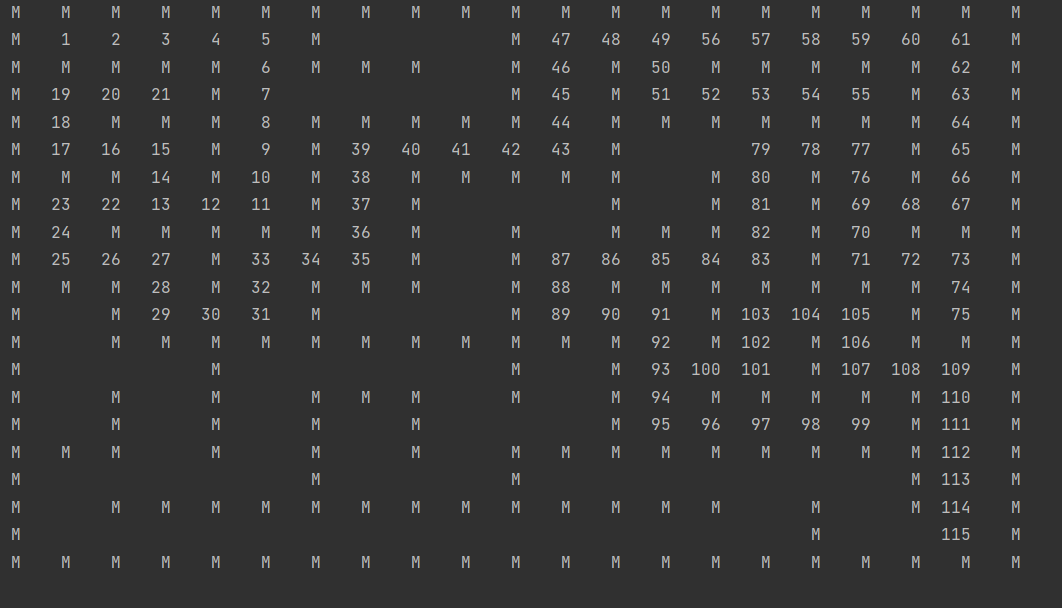
На рисунках 3.1 і 3.2 показані приклади роботи програми для різних алгоритмів пошуку.

Рисунок 3.1 – Алгоритм LDFS

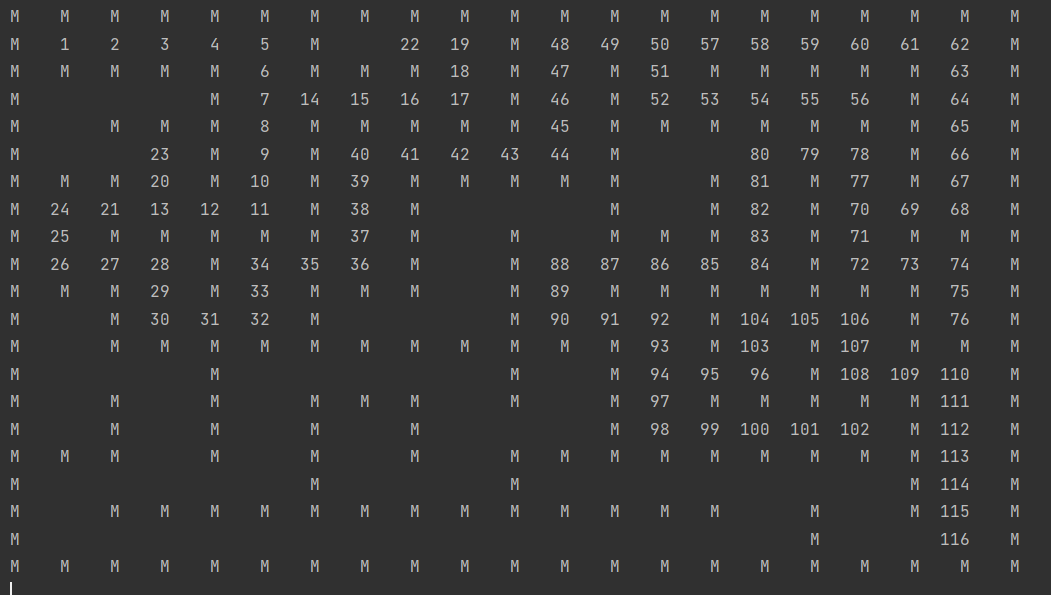


Рисунок 3.2 – Алгоритм RBFS

## Дослідження алгоритмів

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

Таблиця 3.1 – Характеристики оцінювання алгоритму LDFS

|  |  |  |  |
| --- | --- | --- | --- |
| Початкові стани | Ітерації | Всього станів | Всього станів у пом’яті |
| Стан 1 | 115 | 118 | 5 |
| Стан 2 | 119 | 124 | 6 |
| Стан 3 | 37 | 38 | 2 |
| Стан 4 | 61 | 62 | 3 |
| Стан 5 | 147 | 150 | 7 |
| Стан 6 | 103 | 104 | 4 |
| Стан 7 | 95 | 97 | 4 |
| Стан 8 | 73 | 76 | 4 |
| Стан 9 | 115 | 117 | 4 |
| Стан 10 | 149 | 150 | 6 |
| Стан 11 | 145 | 146 | 5 |
| Стан 12 | 65 | 65 | 2 |
| Стан 13 | 73 | 79 | 7 |
| Стан 14 | 65 | 66 | 3 |
| Стан 15 | 113 | 115 | 4 |
| Стан 16 | 75 | 79 | 5 |
| Стан 17 | 103 | 106 | 4 |
| Стан 18 | 103 | 107 | 5 |
| Стан 19 | 135 | 138 | 5 |
| Стан 20 | 51 | 53 | 4 |
| Середнє значення | 97,1 | 99,5 | 4,45 |

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

Таблиця 3.3 – Характеристики оцінювання алгоритму RBFS

|  |  |  |  |
| --- | --- | --- | --- |
| Початкові стани | Ітерації | Всього станів | Всього станів у пом’яті |
| Стан 1 | 116 | 120 | 6 |
| Стан 2 | 139 | 144 | 6 |
| Стан 3 | 37 | 38 | 2 |
| Стан 4 | 45 | 48 | 4 |
| Стан 5 | 123 | 127 | 7 |
| Стан 6 | 92 | 94 | 4 |
| Стан 7 | 93 | 96 | 5 |
| Стан 8 | 68 | 72 | 5 |
| Стан 9 | 70 | 73 | 4 |
| Стан 10 | 107 | 110 | 4 |
| Стан 11 | 128 | 131 | 6 |
| Стан 12 | 61 | 62 | 3 |
| Стан 13 | 109 | 114 | 6 |
| Стан 14 | 58 | 60 | 4 |
| Стан 15 | 113 | 115 | 4 |
| Стан 16 | 75 | 79 | 5 |
| Стан 17 | 105 | 108 | 4 |
| Стан 18 | 132 | 136 | 5 |
| Стан 19 | 101 | 106 | 6 |
| Стан 20 | 79 | 80 | 3 |
| Середнє значення | 92,55 | 95,65 | 4,65 |

Дослідження проведено для однакових початкових станів.

Висновок

При виконанні даної лабораторної роботи було розглянуто алгоритми пошуку в глибину з обмеженням та рекурсивного пошуку за першим найкращим збігом на прикладі задачі з пошуку шляху в лабіринті. Величиною, за якою обирав шлях другий алгоритм стала Евклідова відстань від поточної точки в лабіринті до бажаної. Чим меншою була ця відстань, тим більший пріоритет надавав алгоритм цьому шляху під час пошуку. Шляхом досліджень було встановлено, що RBFS не є обов’язково більш ефективним алгоритмом для пошуку шляху в одиничному випадку, однак на відстані показує кращі результати ніж LDFS. З точки зору використованої пам’яті більш ефективним є LDFS, однак різниця не є значною.

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%.