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) *ІП-15 Волинець Кирило Михайлович* _____
(шифр, прізвище, ім'я, по батькові)

Перевірив *Головченко М.М.* _____
(прізвище, ім'я, по батькові)

Київ 2022

ЗМІСТ

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 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:

- **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	АІП	АЛП	Func
5	Лабіринт	IDS	A*		H2

3 ВИКОНАННЯ

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 neightbor in start.neightbors do
        result = DFS(neightbor, 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
        findedIndex = min_element(priorityQueue)
        result = findAnyWay(priorityQueue, visitedTable, findedIndex,
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
        neightbors = finded.getNeightbors
        for neightbor in neightbors do
            if (visited[neightbor.y][neightbor.x]) continue
            queue.push_back(neightbor)
        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

neightbors = finded.getNeightbors

for neightbor in neightbors **do**

if (visited[neightbor.y][neightbor.x]) **continue**

queue.push_back(neightbor)

end for

return false

3.2 Програмна реалізація

3.2.1 Вихідний код

```
#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;
	}


#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)));
		findAnyWay(priorityQueue, visitedTable, 0, endX, endY);
		priorityQueue.erase(priorityQueue.begin());

		bool result = false;
		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;
		}
```

3.2.2 Приклади роботи

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

Рисунок 3.1 – Алгоритм IDS

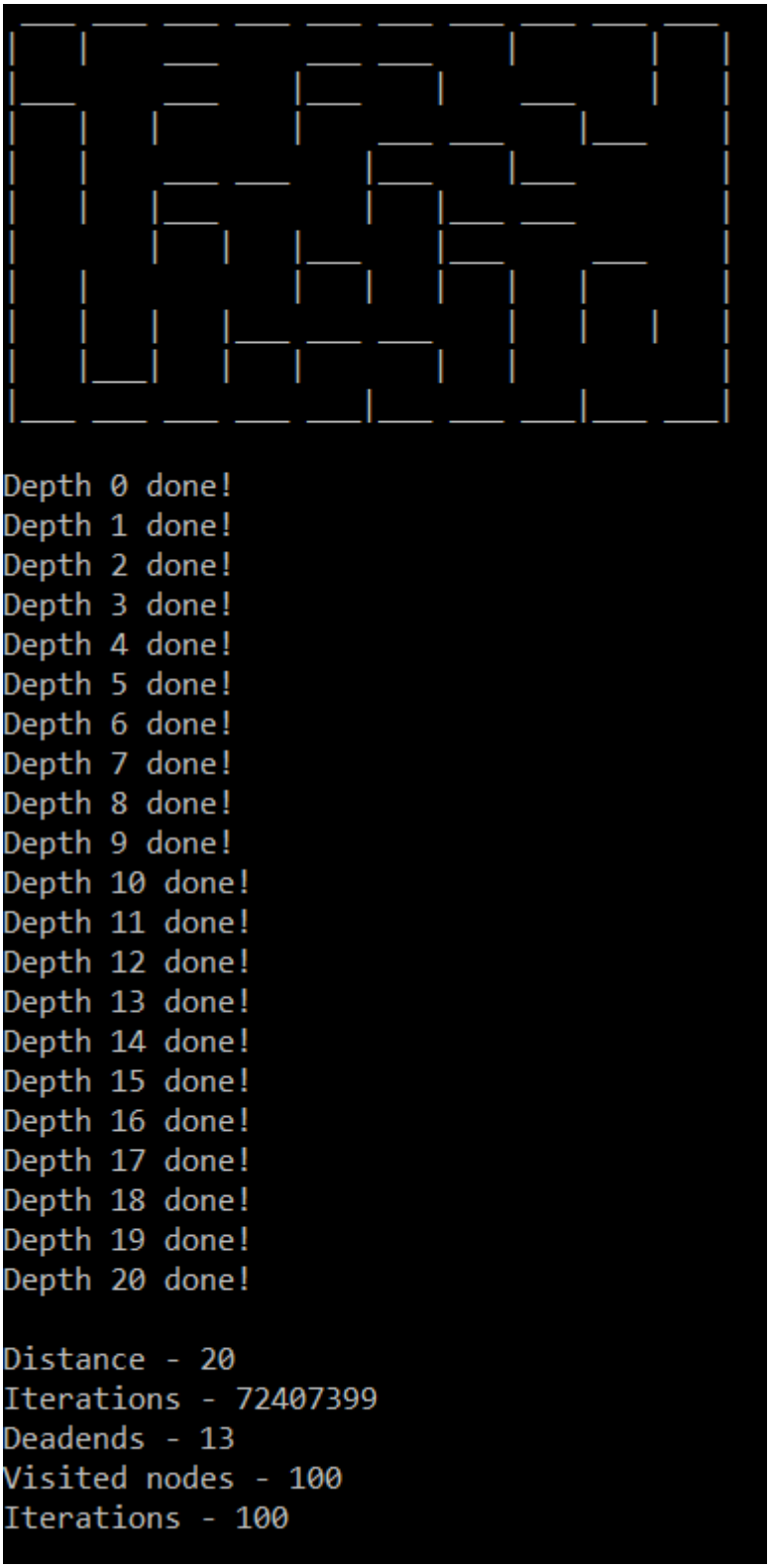
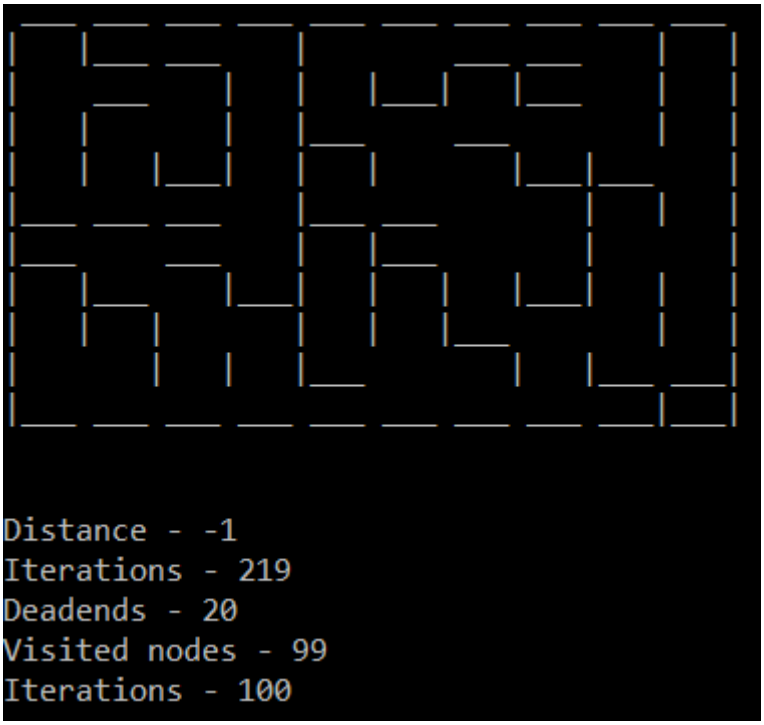


Рисунок 3.2 – Алгоритм A*



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	Перевірених 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 A*

Початкові стани	Шлях	Перевірок клітинки	К-сть гл. кутів	Перевірених 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 A* 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. Досліджені 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%.