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 Гордієнко Ярослав Вадимович* _____
(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.

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 A*.
- **RBFS** – Рекурсивний пошук за першим найкращим співпадінням.
- **F1** – кількість пар ферзів, які б’ють один одного з урахуванням видимості (ферзь A може стояти на одній лінії з ферзем B, проте між ними стоїть ферзь C; тому A не б’є B).
- **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 T 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

3 ВИКОНАННЯ

3.1 Псевдокод алгоритмів

IDS

function Iterative-Deepening-Search(problem) **returns** result or failure

inputs: problem

for depth <= 0 **to** ∞ **do**

result <= Depth-Limited-Search(problem, depth)

if result != cutoff **then return** result

function Depth-Limited-Search (problem, limit) **returns** success or failure/cutoff

return Recursive-DLS(Make-Node(Initial-State[problem]), Problem, limit)

function REcursive-DLS(node, problem, limit) **returns** success or failure/cutoff

if Goal-Test[problem](State[node]) **then return** Solution(node)

else if Deptbh[node] = limit **then return** cutoff

else for each successor in Expand(node, problem) **do**

result <= Recursive-DLS(successor, problem, limit)

if result = cutoff **then** cutoff_occured? <= true

else if result != failure **then return** result

if cutoff_occured?

then return cutoff

else return failure

RBFS

function Recurcive-Best-First-Search(problem) **returns** result or failure

    RBFS(problem, Make-Node(Initial-State[problem]), ∞)

function RBFS(problem, node, f_limit) **returns** success or failure, and f_limit

    if Goal-Test[problem](State[node]) **then return** success

    successors <= Expand(node, problem)

    if successors empty?

        then return failure, ∞

    for each s **in** successors **do**

        F[s] <= g(s) + h(s)

    repeat

        best <= the least heuristic in successors

        if f[best] > f_limit **then return** failure, f[best]

        alternative <= the next least successor heuristic

        result, f[best] <= RBFS(problem, best, min(f_limit, alternative))

        if result ≠ failure **then return** result

3.2 Програмна реалізація

3.2.1 Вихідний код

IDS

```
const MARKERS = {
  cutoff: 1,
  failure: 2,
  success: 3,
  deadEnd: 4,
};

const recursiveDLS = function(node, mazeData, limit, analyzeData) {
  const [ curRowNum, curColNum ] = node.coords;
  const [ solRowNum, solColNum ] = mazeData.cellEnd;

  analyzeData.allStates += 1;

  if (curRowNum === solRowNum && curColNum === solColNum) {
    return {
      marker: MARKERS.success,
    }
  } else if (node.depth === limit) {
    return {
      marker: MARKERS.cutoff,
    }
  }

  if (node.children) {
    analyzeData.statesInMemory += node.children.length;
    for (const child of node.children) {
      drawWay(mazeData, node, child);

      const [ chRowNum, chColNum ] = child.coords;
      const childCell = mazeData.grid[chRowNum][chColNum];
      const childResult = recursiveDLS(childCell, mazeData, limit, analyzeData);

      if (childResult.marker === MARKERS.success) {
        return {
          marker: MARKERS.success,
        }
      }

      if (childResult.marker === MARKERS.cutoff) {
        return {
          marker: MARKERS.cutoff,
        }
      }
    }
```

```
    return {
      marker: MARKERS.failure,
    }
  }

  analyzeData.deadEnds += 1;
  return {
    marker: MARKERS.failure,
  }

};

const dls = function(mazeData, limit, analyzeData) {
  const startCoords = mazeData.cellStart;
  const [ startRow, startCol ] = startCoords;
  const startNode = mazeData.grid[startRow][startCol];

  return recursiveDLS(startNode, mazeData, limit, analyzeData);
};

const ids = function(mazeData) {
  const analyzeData = {
    startStates: mazeData.nodesCount,
    iterations: 0,
    deadEnds: 0,
    allStates: 0,
    statesInMemory: 1,
  };

  let depth = 0;
  let result = null;
  do {
    result = dls(mazeData, depth, analyzeData);
    depth += 1;
    analyzeData.iterations += 1;
  } while (result.marker != MARKERS.success && result.marker != MARKERS.failure)

  showMaze(mazeData);
  console.table(analyzeData);

  return {
    result,
    analyzeData,
  };
};
```

RBFS

```
const MARKERS = {
  failure: 1,
  success: 2,
};

const rbfsRecursive = (mazeData, node, limit, analyzeData) => {
  analyzeData.iterations += 1;

  let succesors = [];

  const [ curRowNum, curColNum ] = node.coords;
  const [ solRowNum, solColNum ] = mazeData.cellEnd;

  if (!node.uniqVisited) {
    analyzeData.allStates += 1;
  }
  node.uniqVisited = 1;

  if (curRowNum === solRowNum && curColNum === solColNum) {
    return {
      marker: MARKERS.success,
    }
  }

  if (!node.children) {
    analyzeData.deadEnds += 1;

    return {
      marker: MARKERS.failure,
      limit: Infinity,
    }
  }

  for (const child of node.children) {
    const [ chRowNum, chColNum ] = child.coords;
    const childCell = mazeData.grid[chRowNum][chColNum];

    const heuristicValue = heuristic(mazeData, childCell);

    succesors.push({
      heuristicValue: heuristicValue,
      cell: childCell,
      childInfo: child,
    });

    analyzeData.statesInMemory += 1;
  }
```

```
  while(true) {
    succesors.sort((a, b) => a.heuristicValue - b.heuristicValue);
    const best = succesors[0];

    if (best.heuristicValue > limit) {
      return {
        marker: MARKERS.failure,
        limit: best.heuristicValue,
      };
    }

    const bestCell = best.cell;

    const alternativeSuccesor = succesors[1];

    let newLimit = 0;
    if (alternativeSuccesor) {
      const alternativeLimit = alternativeSuccesor.heuristicValue;
      newLimit = Math.min(alternativeLimit, limit);
    } else {
      newLimit = limit;
    }

    drawWay(mazeData, node, best.childInfo);
    const result = rbfsRecursive(mazeData, bestCell, newLimit, analyzeData);
    if (result.marker !== MARKERS.failure) {
      return result;
    }

    best.heuristicValue = result.limit;

    const wrongWay = succesors.filter((x) => x.heuristicValue != Infinity);
    if (wrongWay.length === 0) {
      return {
        marker: MARKERS.failure,
        limit: Infinity,
      };
    }
  }
};

const rbfs = (mazeData) => {
  const analyzeData = {
    startStates: mazeData.nodesCount,
    iterations: 0,
    deadEnds: 0,
    allStates: 0,
    statesInMemory: 1,
  };
```

```
  const [ startRowNum, startColNum ] = mazeData.cellStart;
  const startNode = mazeData.grid[startRowNum][startColNum];

  const result = rbfsRecursive(mazeData, startNode, Infinity, analyzeData);

  showMaze(mazeData);
  console.table(analyzeData);

  return {
    result,
    analyzeData,
  };
};
```

3.2.2 Приклади роботи

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

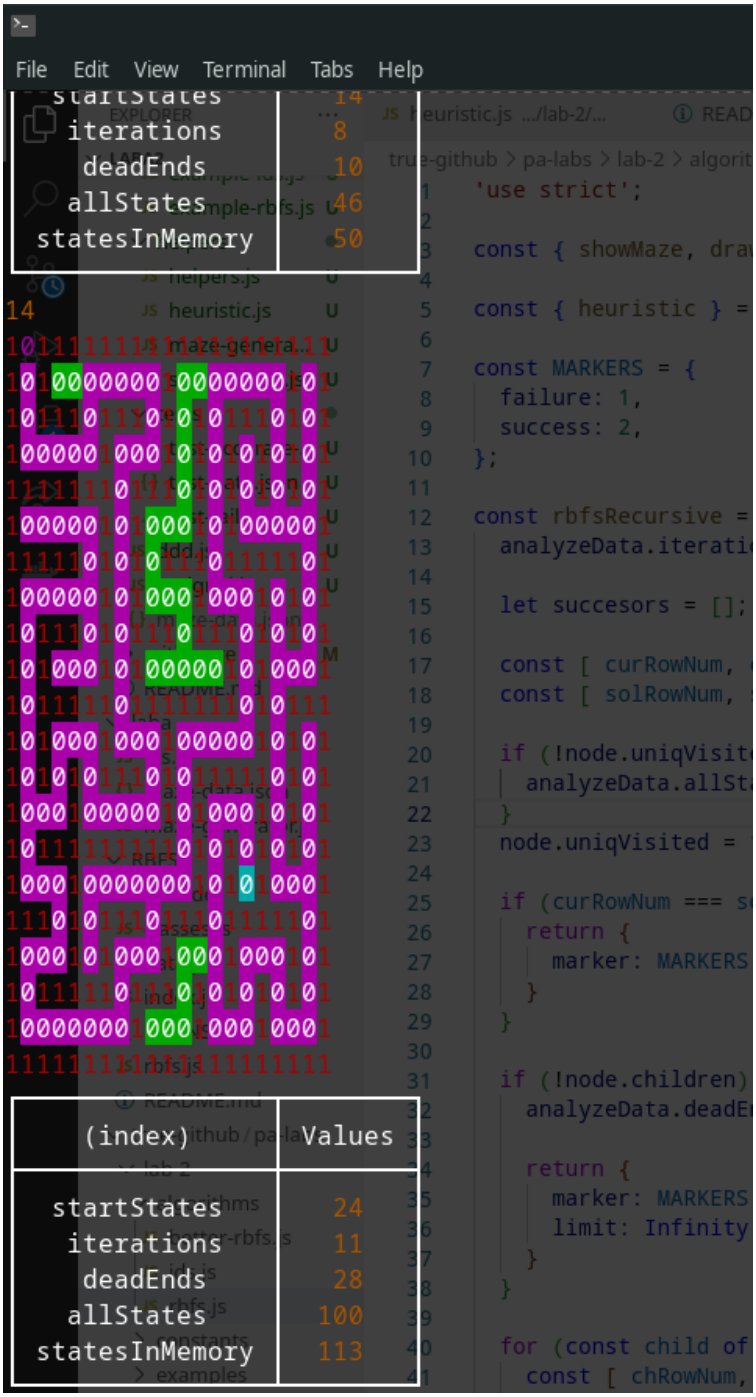

Рисунок 3.1 – Алгоритм IDS

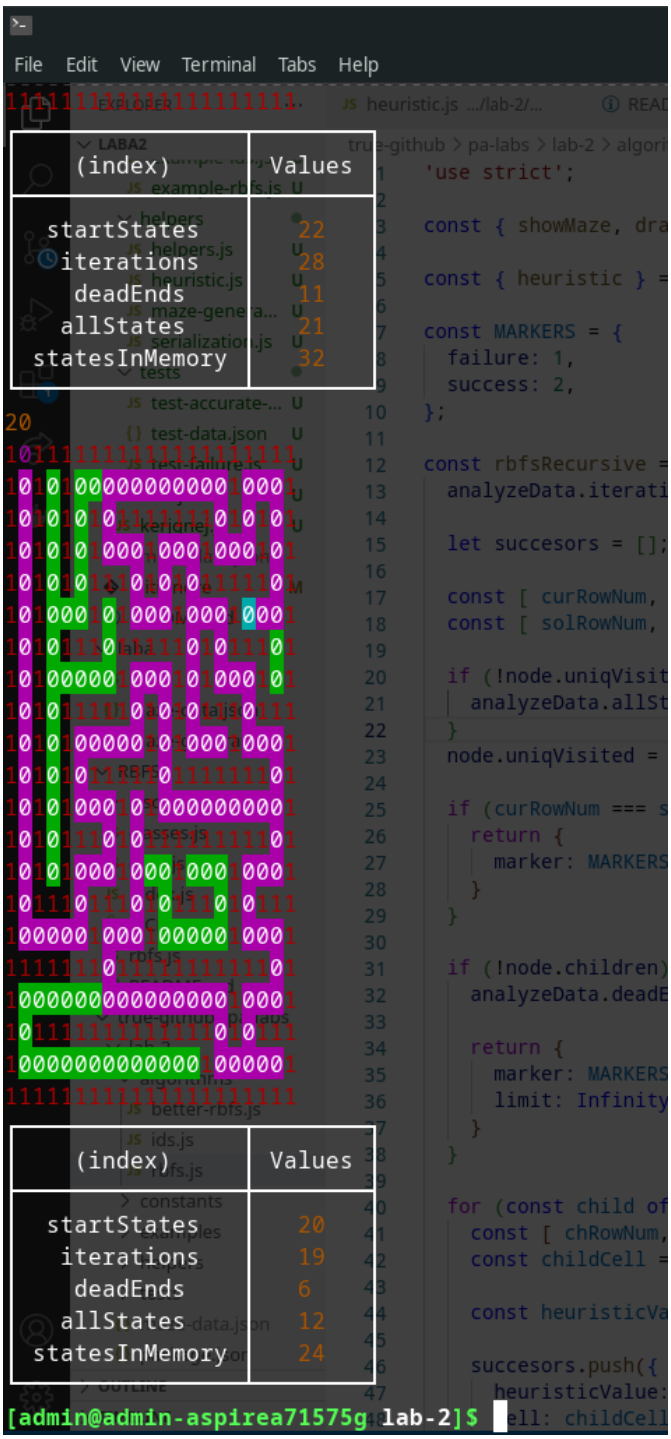

Рисунок 3.2 – Алгоритм RBFS

3.3 Дослідження алгоритмів

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

Таблиця 3.1 – Характеристики оцінювання алгоритму IDS

Початкові стани	Ітерації	К-сть гл. кутів	Всього станів	Всього станів у пам’яті
Стан 1	12	13	91	122
Стан 2	9	18	67	73
Стан 3	10	5	60	82
Стан 4	10	9	64	84
Стан 5	10	21	79	88
Стан 6	11	33	111	125
Стан 7	11	31	101	109
Стан 8	11	19	93	117
Стан 9	13	19	109	143
Стан 10	9	21	73	82
Стан 11	9	12	56	67
Стан 12	13	52	155	169
Стан 13	9	10	55	65
Стан 14	11	26	93	103
Стан 15	14	45	149	168
Стан 16	9	17	64	69
Стан 17	14	48	167	198
Стан 18	10	13	76	98
Стан 19	12	41	128	140
Стан 20	11	23	93	109
Середнє	10.9	23.8	94.2	110.55

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	К-сть гл. кутів	Всього станів	Всього станів у пам’яті
Стан 1	40	18	18	42
Стан 2	20	9	19	21
Стан 3	27	11	21	30
Стан 4	34	14	22	38
Стан 5	17	7	17	18
Стан 6	37	16	21	42
Стан 7	15	5	14	18
Стан 8	28	11	21	32
Стан 9	24	10	17	26
Стан 10	22	5	13	32
Стан 11	19	6	17	24
Стан 12	33	13	20	38
Стан 13	13	4	11	17
Стан 14	23	7	12	30
Стан 15	24	9	17	28
Стан 16	19	9	19	19
Стан 17	19	8	17	20
Стан 18	33	13	23	38
Стан 19	24	10	21	26
Стан 20	24	9	19	28
Середнє	24.75	9.7	17.95	28.35

ВИСНОВОК

При виконанні даної лабораторної роботи було розглянуто алгоритм неінформативного пошуку IDS, а також алгоритм інформативного пошуку RBFS. Загалом з результатів видно, що для не дуже складних за своєю структурою лабіринтів згенерованих алгоритмом DFS, які використовувалися в даній роботі, класичний алгоритм RBFS не видає кардинально кращих за IDS результатів. Це пов’язано з тим, що залежно від початкового стану може скластися ситуація коли алгоритму доводиться вертатися досить далеко для встановлення нової границі. Тому доцільно модифікувати алгоритм RBFS шляхом фіксування нових евристичних значень, при поверненні назад, безпосередньо в структурі даних з лабіринтом або в певному кеші. У багатьох випадках така модифікація здатна суттєво знизити кількість ітерацій алгоритму. У репозиторії лабораторної роботи була розміщена така модифікація.

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%.