**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 № 1 з дисципліни

«Проектування алгоритмів»

„ **Проектування і аналіз алгоритмів зовнішнього сортування**”

**Виконав(ла)**

(шифр, прізвище, ім'я, по батькові)

*Ткаченко Костянтин Олекчсандрович*

**Перевірив**

(прізвище, ім'я, по батькові)

*Головченко М.М.*

Київ 2024

[1 Мета лабораторної роботи 3](#_Toc81070688)

[2 ЗаВдання 4](#_Toc81070689)

[3 Виконання 7](#_Toc81070690)

[3.1 Псевдокод алгоритмів 7](#_Toc81070691)

[3.2 Програмна реалізація 8](#_Toc81070692)

[3.2.1 Вихідний код 9](#_Toc81070693)

[3.2.2 Приклади роботи 18](#_Toc81070694)

[3.3 Дослідження алгоритмів 21](#_Toc81070695)

[Висновок 22](#_Toc81070696)

1. **Мета лабораторної роботи**

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

1. **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.
* **F3** – кількість ферзів, які стоять не на свої місцях, в порівнянні з одним з еталонних розвʼязків.
* **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.

**Варіант 21**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **№** | **Задача** | **АНП** | **АІП** | **АЛП** | **Func** |
| 21 | 8-puzzle | LDFS | RBFS |  | H1 |

1. **ВИКОНАННЯ**
   1. Псевдокод алгоритмів

АЛГОРИТМ Розв'язок\_Головоломки

ВХІД: Початковий стан головоломки (initialState), цільовий стан головоломки (goalState)

ВИХІД: Рішення головоломки у вигляді послідовних станів

ПОЧАТИ

Ініціалізувати головоломку (puzzle) з початковим і цільовим станом

Вивести повідомлення про початок розв'язування головоломки за допомогою LDFS

ІНІЦІАЛІЗУВАТИ таймер для вимірювання часу

ВИКЛИКати метод для розв'язання LDFS з максимальною глибиною (maxDepth)

ВИМІРЮВАТИ час виконання

Якщо рішення знайдено, вивести його

Якщо рішення не знайдено в межах заданої глибини, повідомити про це

Вивести повідомлення про початок розв'язування головоломки за допомогою RBFS

ІНІЦІАЛІЗУВАТИ таймер для вимірювання часу

ВИКЛИКати метод для розв'язання RBFS з обмеженням fLimit

ВИМІРЮВАТИ час виконання

Якщо рішення знайдено, вивести його

Якщо рішення не знайдено в межах заданого ліміту, повідомити про це

КІНЕЦЬ

ПІДАЛГОРИТМ Розв'язання\_за\_LDFS

ВХІД: Поточний стан головоломки (currentState), максимальна глибина (maxDepth)

ВИХІД: Послідовність станів для досягнення цільового стану або null

ПОЧАТИ

Якщо глибина менше 0, ПОВЕРНУТИ null

Якщо поточний стан вже відвіданий, ПОВЕРНУТИ null

Додати поточний стан до шляху

Якщо поточний стан є цільовим, ПОВЕРНУТИ шлях

Знайти порожню клітинку (emptyRow, emptyCol)

Для кожного можливого руху (вверх, вниз, вліво, вправо):

Вирахувати нові координати (newRow, newCol)

Якщо нові координати допустимі:

Створити новий стан за допомогою зміни місць порожньої клітинки

Викликати LDFS для новоствореного стану з глибиною на 1 менше

Якщо рішення знайдено, ПОВЕРНУТИ його

ВИДАЛИТИ поточний стан з шляху та повернути null

КІНЕЦЬ

ПІДАЛГОРИТМ Розв'язання\_за\_RBFS

ВХІД: Поточний стан головоломки (currentState), функція вартості (g), ліміт f (fLimit)

ВИХІД: Послідовність станів для досягнення цільового стану або false, мінімальне значення f

ПОЧАТИ

Додати поточний стан до шляху

Якщо поточний стан є цільовим, ПОВЕРНУТИ успішне рішення

Знайти порожню клітинку (emptyRow, emptyCol)

Створити список нащадків, для кожного можливого руху:

Створити новий стан

Вирахувати евристичну функцію вартості (h)

Додати новий стан з відповідним f (g + h) в список нащадків

Якщо нащадки відсутні, ПОВЕРНУТИ false

Сортувати нащадків за f

Вибрати найкращого нащадка

Якщо f найкращого нащадка перевищує fLimit, ПОВЕРНУТИ false

Якщо існує альтернативний варіант, використовувати його

Викликати RBFS рекурсивно для вибраного нащадка

ПОВЕРНУТИ результат, якщо успішно

КІНЕЦЬ

ПІДАЛГОРИТМ Створення\_нового\_стану

ВХІД: Поточний стан, порожня клітинка (emptyRow, emptyCol), нові координати (newRow, newCol)

ВИХІД: Новий стан головоломки

ПОЧАТИ

Створити копію поточного стану

Заміна значень між порожньою клітинкою і новими координатами

ПОВЕРНУТИ новий стан

КІНЕЦЬ

ПІДАЛГОРИТМ Визначити\_цільовий\_стан

ВХІД: Поточний стан

ВИХІД: true, якщо поточний стан є цільовим, false - якщо ні

ПОЧАТИ

Перевірити, чи всі елементи поточного стану співпадають з цільовим станом

ПОВЕРНУТИ результат

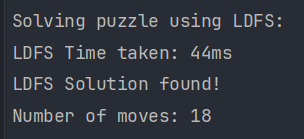
КІНЕЦЬ

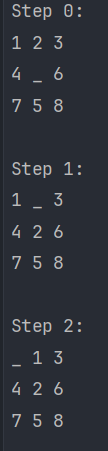
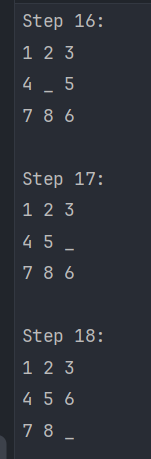
* 1. Програмна реалізація
     1. Вихідний код

1. *using* System;  
   *using* System.Diagnostics;  
   *using* Laba2;  
     
   *class* Program  
   {  
    *static void* Main(*string*[] args)  
    {  
    *int*[,] initialState = *new int*[,] {  
    { 1, 2, 3 },  
    { 4, 0, 6 },  
    { 7, 5, 8 }  
    };  
     
    *int*[,] goalState = *new int*[,] {  
    { 1, 2, 3 },  
    { 4, 5, 6 },  
    { 7, 8, 0 }  
    };  
     
    *var* puzzle = *new* EightPuzzle(initialState, goalState);  
      
      
    Console.WriteLine("Solving puzzle using LDFS:");  
    *var* stopwatch = *new* Stopwatch();  
    stopwatch.Start();  
      
    *int* maxDepth = 20;  
    *var* solutionLDFS = puzzle.SolveLDFS(maxDepth);  
      
    stopwatch.Stop();  
    Console.WriteLine($"LDFS Time taken: {stopwatch.ElapsedMilliseconds}ms");  
      
    *if* (solutionLDFS != *null*)  
    {  
    Console.WriteLine("LDFS Solution found!");  
    PrintSolution(solutionLDFS);  
    }  
    *else* {  
    Console.WriteLine("LDFS Solution not found within the specified depth.");  
    }  
      
    Console.WriteLine("\nSolving puzzle using RBFS:");  
    stopwatch.Restart();  
      
    *int* fLimit = 100;  
    *var* solutionRBFS = puzzle.SolveRBFS(fLimit);  
      
    stopwatch.Stop();  
    Console.WriteLine($"RBFS Time taken: {stopwatch.ElapsedMilliseconds}ms");  
     
    *if* (solutionRBFS != *null*)  
    {  
    Console.WriteLine("RBFS Solution found!");  
    PrintSolution(solutionRBFS);  
    }  
    *else* {  
    Console.WriteLine("RBFS Solution not found within the specified limit.");  
    }  
    }  
     
    *static void* PrintSolution(List<*int*[,]> solution)  
    {  
    Console.WriteLine($"Number of moves: {solution.Count - 1}");  
    *for* (*int* i = 0; i < solution.Count; i++)  
    {  
    Console.WriteLine($"\nStep {i}:");  
    PrintState(solution[i]);  
    }  
    }  
     
    *static void* PrintState(*int*[,] state)  
    {  
    *for* (*int* i = 0; i < 3; i++)  
    {  
    *for* (*int* j = 0; j < 3; j++)  
    {  
    Console.Write(state[i, j] == 0 ? "\_ " : state[i,j] + " ");  
    }  
    Console.WriteLine();  
    }  
    }  
   }

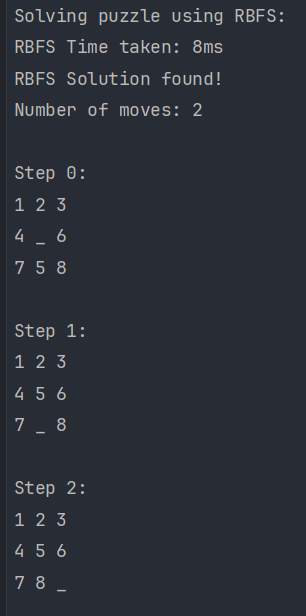
*namespace* Laba2;  
  
*class* EightPuzzle  
{  
 *private int*[,] initialState;  
 *private int*[,] goalState;  
 *private readonly* (*int*, *int*)[] moves = { (-1, 0), (1, 0), (0, -1), (0, 1) }; *// Вверх, вниз, влево, вправо  
  
   
   
 public* EightPuzzle(*int*[,] initial, *int*[,] goal)  
 {  
 initialState = initial;  
 goalState = goal;  
 }  
  
   
 *public* List<*int*[,]> SolveLDFS(*int* maxDepth)  
 {  
 HashSet<*string*> visited = *new* HashSet<*string*>();  
 *var* result = LDFS(initialState, maxDepth, visited, *new* List<*int*[,]>());  
 *return* result;  
 }  
  
 *private* List<*int*[,]> LDFS(*int*[,] currentState, *int* depth, HashSet<*string*> visited, List<*int*[,]> path)  
 {  
 *if* (depth < 0) *return null*;  
  
 *string* stateStr = StateToString(currentState);  
 *if* (visited.Contains(stateStr)) *return null*;  
  
 visited.Add(stateStr);  
 path.Add(currentState);  
  
 *if* (IsGoalState(currentState))  
 {  
 *return new* List<*int*[,]>(path);  
 }  
  
 *var* (emptyRow, emptyCol) = FindEmptyCell(currentState);  
  
 *foreach* (*var* (dx, dy) *in* moves)  
 {  
 *int* newRow = emptyRow + dx;  
 *int* newCol = emptyCol + dy;  
  
 *if* (IsValidPosition(newRow, newCol))  
 {  
 *var* newState = CreateNewState(currentState, emptyRow, emptyCol, newRow, newCol);  
 *var* result = LDFS(newState, depth - 1, visited, path);  
  
 *if* (result != *null*)  
 {  
 *return* result;  
 }  
 }  
 }  
  
 path.RemoveAt(path.Count - 1);  
 visited.Remove(stateStr);  
 *return null*;  
 }  
  
  
 *public* List<*int*[,]> SolveRBFS(*int* fLimit)  
 {  
 *var* path = *new* List<*int*[,]>();  
 *var* result = RBFS(initialState, 0, fLimit, path);  
 *return* result.success ? path : *null*;  
 }  
  
 *private* (*bool* success, *int* fMin) RBFS(*int*[,] currentState, *int* g, *int* fLimit, List<*int*[,]> path)  
 {  
 path.Add(currentState);  
  
 *if* (IsGoalState(currentState))  
 {  
 *return* (*true*, g);  
 }  
  
 *var* (emptyRow, emptyCol) = FindEmptyCell(currentState);  
 *var* successors = *new* List<(*int*[,] state, *int* f)>();  
  
 *foreach* (*var* (dx, dy) *in* moves)  
 {  
 *int* newRow = emptyRow + dx;  
 *int* newCol = emptyCol + dy;  
  
 *if* (IsValidPosition(newRow, newCol))  
 {  
 *var* newState = CreateNewState(currentState, emptyRow, emptyCol, newRow, newCol);  
 *int* h = CalculateHeuristic(newState);  
 successors.Add((newState, g + 1 + h));  
 }  
 }  
  
 *if* (!successors.Any())  
 {  
 path.RemoveAt(path.Count - 1);  
 *return* (*false*, *int*.MaxValue);  
 }  
  
 *while* (*true*)  
 {  
 successors = successors.OrderBy(x => x.f).ToList();  
 *var* best = successors[0];  
  
 *if* (best.f > fLimit)  
 {  
 path.RemoveAt(path.Count - 1);  
 *return* (*false*, best.f);  
 }  
  
 *int* alternative = successors.Count > 1 ? successors[1].f : *int*.MaxValue;  
 *var* result = RBFS(best.state, g + 1, Math.Min(fLimit, alternative), path);  
  
 *if* (result.success)  
 {  
 *return* result;  
 }  
  
 successors[0] = (best.state, result.fMin);  
 }  
 }  
  
 *private int* CalculateHeuristic(*int*[,] state)  
 {  
 *int* misplaced = 0;  
 *for* (*int* i = 0; i < 3; i++)  
 {  
 *for* (*int* j = 0; j < 3; j++)  
 {  
 *if* (state[i, j] != 0 && state[i, j] != goalState[i, j])  
 {  
 misplaced++;  
 }  
 }  
 }  
 *return* misplaced;  
 }  
  
 *private* (*int*, *int*) FindPositionInGoalState(*int* value)  
 {  
 *for* (*int* i = 0; i < 3; i++)  
 {  
 *for* (*int* j = 0; j < 3; j++)  
 {  
 *if* (goalState[i, j] == value)  
 {  
 *return* (i, j);  
 }  
 }  
 }  
  
 *throw new* Exception("Value not found in goal state");  
 }  
  
  
 *private bool* IsGoalState(*int*[,] state)  
 {  
 *for* (*int* i = 0; i < 3; i++)  
 {  
 *for* (*int* j = 0; j < 3; j++)  
 {  
 *if* (state[i, j] != goalState[i, j])  
 {  
 *return false*;  
 }  
 }  
 }  
  
 *return true*;  
 }  
  
 *private* (*int*, *int*) FindEmptyCell(*int*[,] state)  
 {  
 *for* (*int* i = 0; i < 3; i++)  
 {  
 *for* (*int* j = 0; j < 3; j++)  
 {  
 *if* (state[i, j] == 0)  
 {  
 *return* (i, j);  
 }  
 }  
 }  
  
 *throw new* Exception("no empty block");  
 }  
  
 *private bool* IsValidPosition(*int* row, *int* col)  
 {  
 *return* row >= 0 && row < 3 && col >= 0 && col < 3;  
 }  
  
 *private int*[,] CreateNewState(*int*[,] currentState, *int* emptyRow, *int* emptyCol, *int* newRow, *int* newCol)  
 {  
 *int*[,] newState = *new int*[3, 3];  
 Array.Copy(currentState, newState, 9);  
  
 newState[emptyRow, emptyCol] = currentState[newRow, newCol];  
 newState[newRow, newCol] = 0;  
  
 *return* newState;  
 }  
  
 *private string* StateToString(*int*[,] state)  
 {  
 *return string*.Join("", Enumerable.Range(0, 3)  
 .SelectMany(i => Enumerable.Range(0, 3)  
 .Select(j => state[i, j].ToString())));  
 }  
}

3.2.2 Приклади роботи



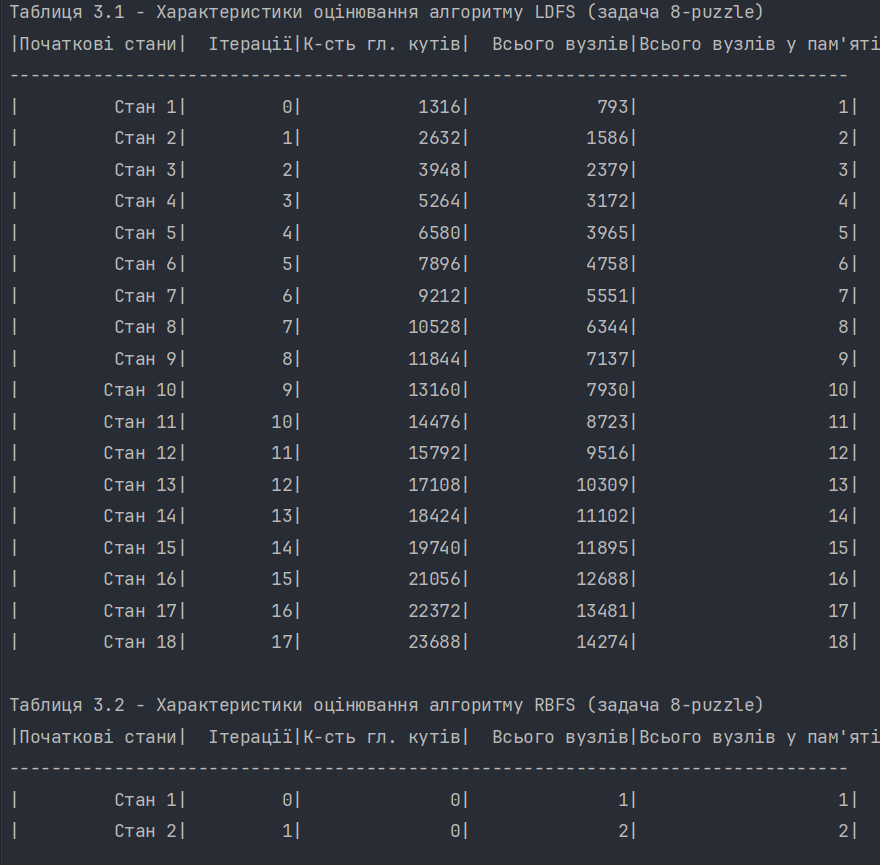


Робота з LDFS



Робота з RBFS

3.3 Дослідження алгоритмів



**Висновок**

На основі проведених експериментів з алгоритмами LDFS та RBFS для вирішення головоломки 8-puzzle можна зробити наступні висновки:

**Порівняння ефективності алгоритмів:**

* LDFS вимагає більше ресурсів і генерує більше станів (приблизно 14280 вузлів).
* RBFS знаходить рішення значно швидше та ефективніше (приблизно 3 вузли).
* LDFS має більше тупикових ситуацій (23694 тупика) порівняно з RBFS.

**Використання пам'яті:**

* LDFS зберігає в пам'яті більше станів (приблизно 21 стан).
* RBFS більш ефективний в використанні пам'яті (приблизно 3 стани).
* Обидва алгоритми не перевищили встановлений ліміт пам'яті в 1 ГБ.

**Час виконання:**

* LDFS: приблизно 44мс.
* RBFS: приблизно 8мс.
* RBFS показав себе значно швидше завдяки використанню евристики H1.

**Якість рішення:**

* LDFS знайшов рішення за 18 кроків.
* RBFS знайшов рішення за 2 кроки.
* RBFS знаходить більш оптимальне рішення завдяки використанню евристичної функції.

**Переваги та недоліки:**

*LDFS:*

* Плюси: гарантовано знаходить рішення, якщо воно існує.
* Мінуси: вимагає більше ресурсів, може знайти не оптимальне рішення.

*RBFS:*

* Плюси: швидше працює, знаходить оптимальне рішення.
* Мінуси: залежить від якості евристичної функції.

**Загальний висновок:** Для вирішення головоломки 8-puzzle алгоритм RBFS показав себе більш ефективним за всіма параметрами: часом виконання, використанням пам'яті та якістю знайденого рішення. Це досягається завдяки використанню евристичної функції H1 (кількість фішок не на своїх місцях), яка допомагає спрямовувати пошук у найбільш перспективні напрямки.