**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-21 Загребельний О. А.*

**Перевірив**

(прізвище, ім'я, по батькові)

*Головченко М.М.*

Київ 2023

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 11](#_Toc81070693)

[3.2.2 Приклади роботи 14](#_Toc81070694)

[3.3 Дослідження алгоритмів 16](#_Toc81070695)

[Висновок 19](#_Toc81070696)

[Критерії оцінювання 2](#_Toc81070697)0

# 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

**BFS**

function BFS\_NQueens(board):

Create a queue data structure and add the initial state (board) to the queue

Create a set to keep track of visited states (closedSet)

while the queue is not empty:

current = dequeue a state from the queue

if current is a goal state (no two queens attack each other):

return current as the solution

add current state to the closedSet

for each child state generated from current state:

if child is not in closedSet:

enqueue child state to the queue

return no solution (if the queue becomes empty)

function GenerateChildren(state):

children = an empty list

for each row in state:

for each column in state:

if placing a queen at (row, column) is a valid move:

create a new state by placing a queen at (row, column)

add the new state to children list

return children

function IsGoalState(state):

for each queen in the state:

for each other queen in the state:

if two queens attack each other:

return false

return true

initialBoard = input board

solution = BFS\_NQueens(initialBoard)

if solution is not null:

print "Solution found:"

print solution

else:

print "No solution found"

**A\***

function AStar\_NQueens(board):

Create an open list to store nodes to be evaluated

Create a closed list to store nodes that have been evaluated

Create the initial node (start) with the given board state

Add the start node to the open list

while the open list is not empty:

current = select the node with the lowest F score from the open list

if current is a goal state (H equals 0, i.e., no queens attack each other):

return current as the solution

Move current from the open list to the closed list

for each child node in current's generated children:

if child is in the closed list:

skip to the next child

if child is not in the open list:

calculate G, H, and F scores for child

set current as the parent of child

add child to the open list

if child is in the open list:

if the G score of child is lower with the current path:

update G, H, and F scores of child

set current as the parent of child

return no solution (if the open list becomes empty)

function GenerateChildren(node):

children = an empty list

for each row in node's state:

for each column in node's state:

if placing a queen at (row, column) is a valid move:

create a new node with the updated state

set current node as the parent of the new node

add the new node to children list

return children

function IsGoalState(node):

Calculate the H score for the node based on the number of attacking queens

if H is 0, return true (goal state)

else, return false

initialBoard = input board

solution = AStar\_NQueens(initialBoard)

if solution is not null:

print "Solution found:"

print solution

else:

print "No solution found"

## Програмна реалізація

### Вихідний код

**BFS.cs**

namespace NonInformativeSearch

{

public class BFS

{

public class Node : DataStructure.Node

{

public Node(int[] state) : base(state) { }

public bool IsGoalState()

{

for (int i = 0; i < this.State.Length; i++)

{

for (int j = i + 1; j < this.State.Length; j++)

{

if (this.State[i] == this.State[j] ||

this.State[i] - i == this.State[j] - j ||

this.State[i] + i == this.State[j] + j)

{

return false;

}

}

}

return true;

}

public new Node[] GenerateChildren()

{

int length = this.State.Length \* (this.State.Length - 1);

Node[] children = new Node[length];

for (int i = 0, index = 0; i < this.State.Length; i++)

{

for (int j = 0; j < this.State.Length; j++)

{

if (this.State[i] != j)

{

int[] newState = new int[this.State.Length];

Array.Copy(this.State, newState, newState.Length);

newState[i] = j;

children[index] = new Node(newState);

index++;

}

}

}

return children;

}

}

public static Results Search(int[] board)

{

Console.WriteLine("\_\_\_.:BFS algorithm:.\_\_\_");

Results results = new Results();

Node start = new Node(board);

Console.WriteLine("Start state:");

Chessboard.PrintResult(start.State);

Queue<Node> openQueue = new Queue<Node>();

HashSet<string> closedSet = new HashSet<string>();

openQueue.Enqueue(start);

while (openQueue.Count > 0)

{

results.Steps++;

Node current = openQueue.Dequeue();

if (current.IsGoalState())

{

results.Board = current.State;

results.StatesInMemory = openQueue.Count + closedSet.Count;

return results;

}

closedSet.Add(string.Join(",", current.State));

foreach (Node child in current.GenerateChildren())

{

if (!closedSet.Contains(string.Join(",", child.State)))

{

results.States++;

openQueue.Enqueue(child);

}

}

}

return null;

}

}

}

**AStar.cs**

namespace InformariveSearch

{

public class AStar

{

public class Node : DataStructure.Node

{

public Node Parent;

public int G;

public int H;

public int F;

public Node(int[] state, Node parent = null) : base(state)

{

this.Parent = parent;

this.G = parent == null ? 0 : parent.G + 1;

this.H = this.CalculateHeuristic();

this.F = this.G + this.H;

}

public int CalculateHeuristic()

{

int count = 0;

for (int i = 0; i < State.Length; i++)

{

for (int j = i + 1; j < State.Length; j++)

{

if (State[i] == State[j] || State[i] - i == State[j] - j || State[i] + i == State[j] + j)

{

count++;

}

}

}

return count;

}

public new Node[] GenerateChildren()

{

int length = this.State.Length \* (this.State.Length - 1);

Node[] children = new Node[length];

for (int i = 0, index = 0; i < this.State.Length; i++)

{

for (int j = 0; j < this.State.Length; j++)

{

if (this.State[i] != j)

{

int[] newState = new int[this.State.Length];

Array.Copy(this.State, newState, newState.Length);

newState[i] = j;

children[index] = new Node(newState, this);

index++;

}

}

}

return children;

}

}

public static Results Search(int[] board)

{

Console.WriteLine("\_\_\_.:A\* algorithm:.\_\_\_");

Results results = new Results();

Node start = new Node(board);

Console.WriteLine("Start state:");

Chessboard.PrintResult(start.State);

List<Node> openList = new List<Node>();

List<Node> closedList = new List<Node>();

openList.Add(start);

while (openList.Count > 0)

{

results.Steps++;

Node current = openList.OrderBy(node => node.F).First();

if (current.H == 0)

{

results.Board = current.State;

results.StatesInMemory = openList.Count + closedList.Count;

return results;

}

openList.Remove(current);

closedList.Add(current);

foreach (Node child in current.GenerateChildren())

{

if (closedList.Any(x => x.State.SequenceEqual(child.State)))

{

continue;

}

if (!openList.Any(x => x.State.SequenceEqual(child.State)))

{

results.States++;

openList.Add(child);

}

else

{

Node existing = openList.First(x => x.State.SequenceEqual(child.State));

if (child.G < existing.G)

{

results.States++;

openList.Remove(existing);

openList.Add(child);

}

}

}

}

return null;

}

}

}

### Приклади роботи

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

Зображення, що містить текст, знімок екрана, Шрифт, дизайн

Автоматично згенерований опис

Рисунок 3.1 – Алгоритм BFS

Зображення, що містить текст, знімок екрана, Шрифт, дизайн

Автоматично згенерований опис

Рисунок 3.2 – Алгоритм A\*

## Дослідження алгоритмів

В таблиці 3.1 наведені характеристики оцінювання алгоритму BFS, задачі 8 ферзів для 20 початкових станів.

Таблиця 3.1 – Характеристики оцінювання алгооритму BFS

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Start states** | **Steps (Iterations)** | **Dead End** | **States** | **States in memory** |
| 3, 3, 2, 3, 4, 1, 6, 5 | - | 1 | - | - |
| 5, 1, 1, 4, 4, 3, 2, 2 | 1079055 | 0 | 34658134 | 33702883 |
| 6, 3, 4, 3, 1, 4, 2, 2 | - | 1 | - | - |
| 6, 4, 6, 3, 2, 6, 3, 4 | - | 1 | - | - |
| 5, 0, 3, 5, 0, 4, 1, 6 | 959656 | 0 | 31054699 | 30208224 |
| 1, 5, 0, 2, 3, 5, 3, 5 | 485187 | 0 | 16674325 | 16250048 |
| 3, 4, 1, 4, 6, 1, 4, 0 | 1405625 | 0 | 44129204 | 42869882 |
| 2, 4, 7, 3, 0, 6, 1, 5 | 52709 | 0 | 2019416 | 1983509 |
| 5, 0, 5, 1, 1, 6, 6, 4 | - | 1 | - | - |
| 5, 5, 1, 0, 6, 0, 1, 5 | - | 1 | - | - |
| 3, 2, 6, 2, 4, 3, 1, 3 | 585907 | 0 | 19892234 | 19380283 |
| 2, 6, 6, 4, 4, 1, 6, 6 | 892913 | 0 | 29052704 | 28270958 |
| 5, 0, 4, 1, 7, 2, 6, 3 | 85212 | 0 | 3133359 | 3068328 |
| 3, 3, 6, 6, 5, 6, 2, 2 | - | 1 | - | - |
| 2, 0, 0, 4, 5, 0, 3, 5 | 50976 | 0 | 1958572 | 1924146 |
| 1, 3, 5, 0, 6, 3, 6, 3 | 36378 | 0 | 1433053 | 1410204 |
| 5, 6, 5, 2, 3, 5, 5, 2 | 497292 | 0 | 17078302 | 16644977 |
| 5, 4, 2, 5, 1, 1, 5, 5 | - | 1 | - | - |
| 2, 2, 0, 1, 0, 3, 1, 3 | 592077 | 0 | 20096119 | 19579482 |
| 3, 2, 6, 1, 6, 5, 0, 2 | 618945 | 0 | 20917304 | 20378185 |
| **Average** | 564764 | 0,35 | 18622878,85 | 18128546,85 |

В таблиці 3.2 наведені характеристики оцінювання алгоритму A\*, задачі 8 ферзів для 20 початкових станів.

Таблиця 3.2 – Характеристики оцінювання А\*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Start states** | **Steps (Iterations)** | **Dead End** | **States** | **States in memory** |
| 3, 3, 2, 3, 4, 1, 6, 5 | 12 | 0 | 533 | 534 |
| 5, 1, 1, 4, 4, 3, 2, 2 | 117 | 0 | 4494 | 4414 |
| 6, 3, 4, 3, 1, 4, 2, 2 | 454 | 0 | 15931 | 15315 |
| 6, 4, 6, 3, 2, 6, 3, 4 | 46 | 0 | 1982 | 1938 |
| 5, 0, 3, 5, 0, 4, 1, 6 | 47 | 0 | 2002 | 1983 |
| 1, 5, 0, 2, 3, 5, 3, 5 | 10 | 0 | 431 | 432 |
| 3, 4, 1, 4, 6, 1, 4, 0 | 12 | 0 | 525 | 526 |
| 2, 4, 7, 3, 0, 6, 1, 5 | 5 | 0 | 201 | 202 |
| 5, 0, 5, 1, 1, 6, 6, 4 | 67 | 0 | 2861 | 2815 |
| 5, 5, 1, 0, 6, 0, 1, 5 | 7 | 0 | 296 | 297 |
| 3, 2, 6, 2, 4, 3, 1, 3 | 14 | 0 | 607 | 599 |
| 2, 6, 6, 4, 4, 1, 6, 6 | 5 | 0 | 201 | 202 |
| 5, 0, 4, 1, 7, 2, 6, 3 | 15 | 0 | 631 | 631 |
| 3, 3, 6, 6, 5, 6, 2, 2 | 10 | 0 | 438 | 439 |
| 2, 0, 0, 4, 5, 0, 3, 5 | 4 | 0 | 153 | 154 |
| 1, 3, 5, 0, 6, 3, 6, 3 | 11 | 0 | 483 | 466 |
| 5, 6, 5, 2, 3, 5, 5, 2 | 6 | 0 | 249 | 250 |
| 5, 4, 2, 5, 1, 1, 5, 5 | 117 | 0 | 4863 | 4697 |
| 2, 2, 0, 1, 0, 3, 1, 3 | 64 | 0 | 2619 | 2547 |
| 3, 2, 6, 1, 6, 5, 0, 2 | 38 | 0 | 1590 | 1591 |
| **Average** | 53,05 | 0 | 2054,5 | 2001,6 |

Висновок

У даній лабораторній роботі були досліджені та реалізовані алгоритми для вирішення задачі вісім ферзів (8-ферзів) за допомогою різних методів пошуку: неінформативного пошуку АНП (BFS), інформативного пошуку АІП (A\*), що використовує задану евристичну функцію.

Під час роботи було створено програму на C#, яка дозволяє вирішувати задачу вісім ферзів за допомогою обраних алгоритмів. Програма дозволяє обмежити максимальний обсяг оперативної пам’яті, генерувати шахову дошку для N ферзів, а також надає можливість вводити початковий стан вручну.

Для оцінки ефективності роботи алгоритмів було проведено серію експериментів з різними початковими станами задачі. Параметри, які було виміряно під час експериментів, включають середню кількість етапів (кроків) для досягнення розв'язку, середню кількість випадків, коли алгоритм не знайшов розв'язку, середню кількість згенерованих станів під час пошуку та середню кількість станів, що зберігаються в пам'яті під час роботи програми.

З результатів експериментів видно, що A\* значно ефективніший за BFS у розв'язанні задачі вісім ферзів. Він використовує набагато менше ітерацій і пам’яті та створює менше станів, внаслідок чого працює значно швидше і для 8-ми ферзів завжди знаходить оптимальний розв’язок задачі, чого не можна сказати про BFS.

У цілому, лабораторна робота дозволила реалізувати та порівняти різні методи пошуку для задачі вісім ферзів, а також вивчити їх ефективність у різних умовах. Результати експериментів можуть бути корисними для вибору оптимального алгоритму при вирішенні подібних задач.

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%.