**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»

**«Неінформативний, інформативний та локальний пошук»**

**Виконав(ла)**

(шифр, прізвище, ім'я, по батькові)

*ІП-01 Галько Міла Вячеславівна*

**Перевірив**

(прізвище, ім'я, по батькові)

*Головченко М.М.*

Київ 2021

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. 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 (512 Мб)

**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.
* **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.
* **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** | **Func** |
| 6 | 8-ферзів | BFS | A\* | F2 |

# Виконання

## Псевдокод алгоритмів

BFS:

BFS(*start\_node*, *goal\_node*) {

**return** BFS'({*start\_node*}, ∅, *goal\_node*);

}

BFS'(*fringe*, *visited*, *goal\_node*) {

**if**(*fringe* == ∅) {

// Целевой узел не найден

**return** false;

}

**if** (*goal\_node* ∈ *fringe*) {

**return** true;

}

**return** BFS'({*child* | *x* ∈ *fringe*, *child* ∈ expand(*x*)} \ *visited*, *visited* ∪ *fringe*, *goal\_node*);

}

A\*:

**bool** A\*(start, goal)**:**

U = ∅∅

Q = ∅∅

Q.push(start)

g[start] = 0

f[start] = g[start] + h(start)

**while** Q.size() != 0

current = вершина из QQ с минимальным значением ff

**if** current == goal

**return** *true* // нашли путь до нужной вершины

Q.remove(current)

U.push(current)

**for** v : смежные с current вершины

tentativeScore = g[current] + d(current, v) // d(current, v) — стоимость пути между current и v

**if** v∈Uv∈U **and** tentativeScore >= g[v]

**continue**

**if** v∉Uv∉U **or** tentativeScore < g[v]

parent[v] = current

g[v] = tentativeScore

f[v] = g[v] + h(v)

**if** v∉Qv∉Q

Q.push(v)

**return** *false*

## Програмна реалізація

### Вихідний код

#include "Algorithm.hpp"

**int** main() {

Board board;

board.GetFirstState("board.txt");

**if** (!board.GoodState()) {

Algorithm algorithm;

algorithm.startAlgo(board);

}

**else** {

cout << "true" << endl;

}

**return** 0;

}

#pragma once

#include "Node.hpp"

**class** Algorithm {

**int** statesNumBFS, statesNumAStar;

**int** iterationBFS, iterationAStar;

**friend** **bool** **operator**<(**const** pair<**int**, Board>&, **const** pair<**int**, Board>&);

**using** pr\_queue = priority\_queue<pair<**int**, Node\*>, vector<pair<**int**, Node\*>>, std::greater<pair<**int**, Node\*>>>;

**bool** BFS(Board);

**bool** AStar(Board);

**void** generateChildrenBFS(queue<Node\*>&, Node\*, set<Node\*>&);

**void** generateChildrenAStar(pr\_queue&, Node\*, set<Node\*>&);

**public**:

Algorithm(): statesNumBFS(0), statesNumAStar(0), iterationBFS(0), iterationAStar(0) {}

**void** startAlgo(Board);

};

#include "Algorithm.hpp"

**void** Algorithm::startAlgo(Board board) {

cout << "First State:" << endl;

board.Output();

cout << endl;

cout << "AStar:" << endl;

**unsigned** **int** start1 = clock();

AStar(board);

**unsigned** **int** end1 = clock();

cout << "Time: " << (**float**)(end1 - start1) / CLOCKS\_PER\_SEC << endl;

cout << endl;

cout << "BFS:" << endl;

**unsigned** **int** start2 = clock();

BFS(board);

**unsigned** **int** end2 = clock();

cout << "Time: " << (**float**)(end2-start2) / CLOCKS\_PER\_SEC << endl;

}

**bool** Algorithm::BFS(Board board) {

Node \*current = **new** Node(board);

statesNumBFS++;

queue<Node\*> nodes;

nodes.push(current);

set<Node\*> visited;

visited.insert(current);

**do** {

current = nodes.front();

nodes.pop();

iterationBFS++;

**if**(current->board.GoodState()) {

current->board.Output();

cout << "Iteration: " << iterationBFS << endl;

cout << "All state's number: " << statesNumBFS << endl;

cout << "States in queue (finally): " << nodes.size() << endl;

**return** **true**;

}

generateChildrenBFS(nodes, current, visited);

} **while** (!nodes.empty());

**return** **false**;

}

**void** Algorithm::generateChildrenBFS(queue<Node\*> &nodes, Node\* node, set<Node\*> &visited) {

Board subBoard;

**for** (**int** row = 0; row < boardSize; row++) {

**for** (**int** column = 0; column < boardSize; column++) {

subBoard.squares[row][column] = node->board.squares[row][column];

}

}

**for** (**int** row = 0; row < boardSize; row++) {

**for** (**int** column = 0; column < boardSize; column++) {

**if** (node->board.squares[row][column] == 1) {

subBoard.squares[row][column] = 0;

**for** (**int** subColumn = 0; subColumn < boardSize; subColumn++) {

**if** (subColumn != column) {

subBoard.squares[row][subColumn] = 1;

Node \*child = **new** Node(subBoard);

**if** (visited.find(child) == visited.end()) {

statesNumBFS++;

nodes.push(child);

}

subBoard.squares[row][subColumn] = 0;

}

}

subBoard.squares[row][column] = 1;

**continue**;

}

}

}

}

**bool** **operator**<(**const** pair<**int**, Board> &board1, **const** pair<**int**, Board> &board2) {

**return** board1.first < board2.first;

}

**bool** Algorithm::AStar(Board board) {

Node \*current = **new** Node(board);

statesNumAStar++;

pr\_queue nodes;

**int** f2 = board.GetF2();

nodes.push(make\_pair(f2, current));

set<Node\*> visited;

**do** {

current = nodes.top().second;

nodes.pop();

visited.insert(current);

iterationAStar++;

**if**(current->board.GoodState()) {

current->board.Output();

cout << "Iteration: " << iterationAStar << endl;

cout << "All state's number: " << statesNumAStar << endl;

cout << "States in queue (finally): " << nodes.size() << endl;

**return** **true**;

}

generateChildrenAStar(nodes, current, visited);

} **while** (!nodes.empty());

**return** **false**;

}

**void** Algorithm::generateChildrenAStar(pr\_queue &nodes, Node \*current, set<Node\*> &visited) {

Board subBoard;

**for** (**int** row = 0; row < boardSize; row++) {

**for** (**int** column = 0; column < boardSize; column++) {

subBoard.squares[row][column] = current->board.squares[row][column];

}

}

**for** (**int** row = 0; row < boardSize; row++) {

**for** (**int** column = 0; column < boardSize; column++) {

**if** (current->board.squares[row][column] == 1) {

subBoard.squares[row][column] = 0;

**for** (**int** subColumn = 0; subColumn < boardSize; subColumn++) {

**if** (subColumn != column) {

subBoard.squares[row][subColumn] = 1;

Node \*child = **new** Node(subBoard);

**if** (visited.find(child) == visited.end()) {

statesNumAStar++;

**int** f2 = subBoard.GetF2();

nodes.push(make\_pair(f2, child));

}

subBoard.squares[row][subColumn] = 0;

}

}

subBoard.squares[row][column] = 1;

**continue**;

}

}

}

}

#pragma once

#include "Board.hpp"

**class** Node {

Board board;

Node(Board board) {

**this**->board = board;

}

**friend** **class** Algorithm;

};

#pragma once

#include "libraries.hpp"

**class** Board {

vector<vector<**int**>> squares;

**public**:

Board();

**void** GetFirstState(string);

**bool** GoodState();

**void** Output();

**int** GetF2();

~Board();

**friend** **class** Algorithm;

};

#include "Board.hpp"

Board::Board() {

squares.resize(boardSize);

**for** (**auto** &cell: squares) {

cell.resize(boardSize);

}

}

Board::~Board(){

**for** (**auto** column: squares) {column.clear();}

squares.clear();

}

//PARSING~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

**void** Board::GetFirstState(string file) {

ifstream fin (file);

string boardStates;

getline(fin, boardStates);

**for** (**auto** &cell: squares) {

**int** position = stoi(boardStates.substr(0,1));

cell[position] = 1;

boardStates.erase(0,1);

}

}

//STATE\_CHECKING~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

**bool** Board::GoodState() {

**for** (**int** row = 0; row < boardSize; row++) {

**for** (**int** column = 0; column < boardSize; column++) {

**if** (squares[row][column] == 1) {

**for** (**int** j = row + 1; j < boardSize; j++) {

**if** (squares[j][column] == 1) {

**return** **false**;}

**int** diagonalUp = column + row - j;

**int** diagonalDown = column + j - row;

**if** (diagonalUp >= 0 && squares[j][diagonalUp] == 1) {**return** **false**;}

**if** (diagonalDown < boardSize && squares[j][diagonalDown] == 1) {**return** **false**;}

}

}

}

}

**return** **true**;

};

//H2~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

**int** Board::GetF2() {

**int** conflicts = 0;

**for** (**int** row = 0; row < boardSize; row++) {

**for** (**int** column = 0; column < boardSize; column++) {

**if** (squares[row][column] == 1) {

**bool** UpConflict = **false**;

**bool** DownConflict = **false**;

**for** (**int** j = row + 1; j < boardSize; j++) {

**int** diagonalUp = column + row - j;

**int** diagonalDown = column + j - row;

**if** (diagonalUp >= 0 && !UpConflict && squares[j][diagonalUp] == 1) {

conflicts++;

UpConflict = **true**;

}

**if** (diagonalDown < boardSize && !DownConflict && squares[j][diagonalDown] == 1) {

conflicts++;

DownConflict = **true**;

}

**if** (UpConflict && DownConflict) {

**break**;

}

}

}

}

}

**return** conflicts;

}

//OUTPUT~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

**void** Board::Output() {

**for** (**int** column = 0; column < boardSize; column++) {

**for** (**int** row = 0; row < boardSize; row++) {

cout << squares[row][column] << " ";

}

cout << endl;

}

cout << endl;

};

#pragma once

#include <stdio.h>

#include <iostream>

#include <fstream>

#include <vector>

#include <string>

#include <queue>

#include <time.h>

#include <set>

**using** **namespace** std;

**const** **int** boardSize = 8;

### Приклади роботи

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

## 

## Дослідження алгоритмів

В таблиці 3.1 наведені характеристики оцінювання алгоритму A\* задачі 8 ферзів для 20 початкових станів.

Таблиця 3.1 – Характеристики оцінювання алгоритму A\*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Початкові стани | Ітерації | Всього станів | Всього станів у пам’яті | Час виконання (сек) |
| 11130714 | 347 | 19376 | 19030 | 0.318 |
| 71204635 | 36 | 1960 | 1925 | 0.035 |
| 15067460 | 993 | 55552 | 54560 | 0.873 |
| 02051563 | 79 | 4369 | 4290 | 0.08 |
| 07174630 | 94 | 5209 | 5115 | 0.094 |
| 15067460 | 16776 | 939401 | 922625 | **14.9424** |
| 35716024 | 0 | 0 | 0 | 0 |
| 31120635 | 4 | 169 | 165 | 0.003 |
| 13025267 | 2103 | 117713 | 115610 | 1.891 |
| 50617275 | 161 | 8961 | 8800 | 0.17 |
| 71437625 | 319 | 17809 | 17490 | 0.333289 |
| 46716714 | 1113 | 62273 | 61160 | 1.074 |
| 53420142 | 15102 | 845657 | 830555 | 13.1502 |
| 25064463 | 935 | 52305 | 51370 | 0.922 |
| 25037460 | 2989 | 167329 | 164340 | **2.795** |
| 51430714 | 859 | 48049 | 47190 | **0.868** |
| 62017460 | 780 | 43625 | 42845 | 0.799 |
| 14722713 | 280 | 15625 | 15345 | 0.288 |
| 35027461 | 1053 | 58913 | 57860 | 1.062 |
| 56140645 | 3403 | 190513 | 187110 | 3.12 |

В таблиці 3.2 наведені характеристики оцінювання алгоритму BFS задачі 8 ферзів для 20 початкових станів.

Таблиця 3.3 – Характеристики оцінювання алгоритму BFS

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Початкові стани | Ітерації | Всього станів | Всього станів у пом’яті | Час виконання (сек) |
| 11130714 | 16205 | 907425 | 891220 | 13.276 |
| 71204635 | 57766 | 3234841 | 3177075 | 45.748 |
| 15067460 | 7248 | 405833 | 398585 | 5.82 |
| 02051563 | 433 | 24193 | 23760 | 0.383 |
| 07174630 | 390543 | 21 870353 | 21 479810 | 403.514 |
| 15067460 | 7248 | 405833 | 398585 | **7.361** |
| 35716024 | 0 | 0 | 0 | 0 |
| 31120635 | 411 | 22961 | 22550 | 0.387 |
| 13025267 | 24079 | 1348369 | 1324290 | 19.508 |
| 50617275 | 62382 | 343337 | 340955 | 50.29 |
| 71437625 | 76934 | 4308249 | 4231315 | 61.243 |
| 46716714 | 217712 | 12191817 | 1197410 | 173.4 |
| 53420142 | 63988 | 3583273 | 3519585 | 51.264 |
| 25064463 | 49106 | 2749881 | 2700775 | 38.906 |
| 25037460 | 976 | 54601 | 53625 | **0.852** |
| 51430714 | 525 | 29345 | 28820 | **0.453** |
| 62017460 | 10098 | 565433 | 555335 | 8.414 |
| 14722713 | 4635 | 259505 | 254870 | 3.878 |
| 35027461 | 10383 | 581393 | 571010 | 8.432 |
| 56140645 | 45357 | 2539937 | 2494580 | 35.89 |

Висновок

При виконанні даної лабораторної роботи було розглянуто та досліджено алгоритми інформативного та не інформативного пошуку – BFS, A\*. Базуючись на даних алгоритмах, вирішила задачу про 8 ферзів. Як початковий стан були використані 20 різних станів, де за умовою 1 ферзь займає позицію на 1-ому стовпці. В ході виконання роботи, були:

* дослідженні псевдокоди алгоритмів;
* розроблена програмна реалізація вирішення задачі;
* проведено 20 тестувань на кількість ітерацій, станів, станів у пам’яті та використаного часу.

Як висновок, можна зазначити, що алгоритм А\* є більш оптимальним для вирішення даної задачі, оскільки з-за евристичної функції пропускає більшість станів із великою кількістю конфліктів. І як раз BFS через свою кількість ітерацій забирає багато часу та пам’яті в ході рішення. Тому іноді дана задача не може бути вирішена через обмежену пам’ять комп’ютера.

Критерії оцінювання

За умови здачі лабораторної роботи до 25.09.2021 включно максимальний бал дорівнює – 5. Після 25.09.2021 максимальний бал дорівнює – 1.

Критерії оцінювання у відсотках від максимального балу:

* псевдокод алгоритму – 10%;
* програмна реалізація алгоритму – 60%;
* дослідження алгоритмів – 25%;
* висновок – 5%.