**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-14 Радзівіло Валерія Артемівна*

**Перевірив**

(прізвище, ім'я, по батькові)

*Головченко М.М.*

Київ 2022

Зміст

[1 Мета лабораторної роботи 3](#_Toc81070688)

[2 ЗаВдання 4](#_Toc81070689)

[3 Виконання 7](#_Toc81070690)

[3.1 Псевдокод алгоритмів 7](#_Toc81070691)

[3.2 Програмна реалізація 9](#_Toc81070692)

[3.2.1 Вихідний код 9](#_Toc81070693)

[3.2.2 Приклади роботи 39](#_Toc81070694)

[3.3 Дослідження алгоритмів 40](#_Toc81070695)

[Висновок 45](#_Toc81070696)

[Критерії оцінювання 46](#_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. 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** |
| 14 | 8-ферзів | BFS | RBFS |  | F2 |

# Виконання

## Псевдокод алгоритмів

1. BFS

answer = check\_every\_child(tree.root.children)

if answer==null:

children\_to\_check = tree.root.children

for node in tree:

check\_every\_child(node.children)

if answer!=null:

return answer;

for child in node.children:

1. RBFS(Tree tree, ArrayList<Node>min\_conflicts, ArrayList<Tree> trees\_in\_mem, int iter, Tree primary\_tree, int counter, ArrayList<Integer> answer)

if answer!=null:

return answer;

if trees\_in\_mem.size > 1 and counter%100==0:

// state heading and go from old heading in new route

Tree new\_tree = trees\_in\_mem.get(0)

Clean trees\_in\_mem

trees\_in\_mem.add(new\_tree)

Node min = min\_conflicts.get(0)

Clean min\_conflicts

min\_conflicts.add(min)

answer=RBFS(new\_tree,min\_conflicts,trees\_in\_mem,iter=1, primary\_tree,counter++,answer)

if answer!=null

return answer

if counter%200=0:

//switch to new heading

clean trees\_in\_mem

clean min\_conflicts

answer=RBFS(primary\_tree,min\_conflicts,trees\_in\_mem,iter =0,primary\_tree,counter+=1,answer)

if answer!=null

return answer

start = choose\_less\_conflict\_in\_list(tree.root.children)

if start!=null and start.amount\_of\_conflicts==0:

// the answer is found

return start

else

if start!=null and (start.getDepth()==0 or start.getConflicts()<= find\_min\_node(min\_conflicts).getConflicts()) ) :

Tree new\_tree = new Tree(start);

trees\_in\_mem.add(new\_tree);

min\_conflicts.add(start);

answer= RBFS(new\_tree,min\_conflicts,trees\_in\_mem,iter+=1, primary\_tree,counter+=1,answer);

if(answer!=null)

//the answer is found

return answer

else if trees\_in\_mem.size()>1:

trees\_in\_mem.remove(last element)

min\_conflicts.remove(last element)

Tree new\_tree = trees\_in\_mem.get(trees\_in\_mem.size()-1)

answer=RBFS(new\_tree,min\_conflicts,trees\_in\_mem,iter-=1, primary\_tree,counter+=1,answer)

if answer!=null

return answer

else:

// time to switch heading

counter =200

answer=RBFS(null,min\_conflicts,trees\_in\_mem,iter-=1, primary\_tree,counter,answer);

if answer!=null

return answer

return answer;

1. F2

int pairs=0;

//check diagonals

for (int i=0; i<n;i++)

for(int j =i+1; j<n;j++)

if(j!=i && Math.abs(array[i]-array[j])==Math.abs(i-j))

pairs++;

//check verticals

for(int i =0; i<n;i++)

int tmp =0;

for(int j =i+1; j<n; j++)

if(array[i]==array[j]&& i!=j)

tmp++;

if (tmp>1)

pairs+=tmp;

if(tmp==1)

pairs++;

## Програмна реалізація

### Вихідний код

Main.java:

import ArrWork.\*;  
import java.util.Objects;  
import java.util.Scanner;  
  
  
public class Main {  
 public static void main(String args[]){  
 int n =8;  
  
 int [] initial\_placement = ArrWork.*make\_rand\_int\_arr*(n);  
 System.*out*.println("Initial task: ");  
 ArrWork.*create\_matr*(initial\_placement,n);  
  
 Scanner myObj = new Scanner(System.*in*);  
 System.*out*.println("Write nuber of type BFS(0)/RBFS(1): ");  
 int method = myObj.nextInt();  
 while (method!=0 && method!=1){  
 System.*out*.println("YOU MUST WRITE 0 OR 1, NOTHING ELSE!");  
 System.*out*.println("Write nuber of type BFS(0)/RBFS(1): ");  
 method = myObj.nextInt();  
 }  
 if(method == 0) {  
 int[] result = make\_n\_queens.*make\_n\_queens*(n, false, initial\_placement, method);  
  
 System.*out*.println("Result BFS: ");  
 ArrWork.*create\_matr*(result, n);  
 System.*out*.println("F2 for BFS: " + F2.*F2*(result, 8));  
 }  
 else if (method==1){  
 int[] result = make\_n\_queens.*make\_n\_queens*(n, false, initial\_placement, 1);  
  
 System.*out*.println("Result RBFS: ");  
 ArrWork.*create\_matr*(result, n);  
 System.*out*.println("F2 for RBFS: " + F2.*F2*(result, 8));  
 }  
  
 System.*out*.println("Do you want to see 2 solution?('yes'/'no'): ");  
 myObj =new Scanner(System.*in*);  
 String answer = myObj.nextLine();  
 if(Objects.*equals*(answer, "yes"))  
 {  
 method = Math.*abs*(method -1);  
 System.*out*.println(method);  
 if(method == 0) {  
 int[] result = make\_n\_queens.*make\_n\_queens*(n, false, initial\_placement, method);  
  
 System.*out*.println("Result BFS: ");  
 ArrWork.*create\_matr*(result, n);  
 System.*out*.println("F2 for BFS: " + F2.*F2*(result, 8));  
 }  
 else if (method==1){  
 int[] result = make\_n\_queens.*make\_n\_queens*(n, false, initial\_placement, 1);  
  
 System.*out*.println("Result RBFS: ");  
 ArrWork.*create\_matr*(result, n);  
 System.*out*.println("F2 for RBFS: " + F2.*F2*(result, 8));  
 }  
 }  
  
 }  
  
  
  
}

make\_n\_queens.java:

package ArrWork;  
  
import java.util.ArrayList;  
  
public class make\_n\_queens {  
 public static int[] make\_n\_queens(int n, boolean isRand, int[] notrand, int method) {  
 int [ ] initial\_placement = new int[n];  
 if(isRand)  
 initial\_placement = ArrWork.*make\_rand\_int\_arr*(n);  
 else  
 initial\_placement =notrand.clone();  
  
 System.*out*.println("Initial task: ");  
 ArrWork.*create\_matr*(initial\_placement,n);  
  
 System.*out*.println("F2: "+F2.*F2*(initial\_placement,8));  
  
 ArrayList<Integer> init\_list = ArrWork.*array\_to\_list*(initial\_placement);  
 Tree tree = new Tree(new Node(init\_list,0));  
 tree.make\_tree(tree);  
 int[] answer = new int[8];  
 if(method==0)  
 {  
 ArrayList<Integer> answer\_bfs = *BFS*(tree);  
 answer = ArrWork.*arrlist\_to\_arr*(answer\_bfs);  
 }  
  
 else {  
 ArrayList<Node> min\_conflicts = new ArrayList<>();  
 ArrayList<Tree> trees\_in\_mem = new ArrayList<>();  
 ArrayList<Integer> answer\_rbfs = null;  
 answer\_rbfs = *RBFS*(tree, min\_conflicts, trees\_in\_mem, 0, tree, 0, answer\_rbfs);  
 answer = ArrWork.*arrlist\_to\_arr*(answer\_rbfs);  
  
 }  
  
 return answer;  
 }  
  
  
 static ArrayList<Integer> BFS(Tree tree)  
 {  
 ArrayList<Integer> answer = new ArrayList<>();  
 answer = *check\_every\_child*(tree.getRoot().getChildren());  
 if(answer==null) {  
  
 ArrayList<Node> children\_to\_check = new ArrayList<>();  
 children\_to\_check = tree.getRoot().getChildren();  
 int iterator = 0;  
 while(iterator<100) {  
 ArrayList<Node> get\_children = new ArrayList<>();  
 for (Node child : children\_to\_check) {  
 answer = *check\_every\_child*(child.getChildren());  
 if(answer!=null)  
 {  
 return answer;  
 }  
 for(Node c: child.getChildren())  
 get\_children.add(c);  
 }  
 children\_to\_check = get\_children;  
 iterator++;  
  
 }  
 }  
  
  
  
 return answer;  
 }  
  
 static ArrayList<Integer> check\_every\_child(ArrayList<Node> children\_to\_check)  
 {  
 ArrayList<Integer> check = null;  
 check = *check\_children*(children\_to\_check);  
 if (check != null) return check;  
 return check;  
 }  
  
 static ArrayList<Integer> check\_children (ArrayList<Node>children)  
 {  
 for(Node child:children)  
 {  
 if(!child.visited) {  
 if (*check\_placement*(child.get\_col\_list())) {  
 System.*out*.println("FOUND");  
 return child.get\_col\_list();  
 }  
 child.setVisited(true);  
 }  
 }  
 return null;  
 }  
  
  
 static boolean check\_placement(ArrayList<Integer>list)  
 {  
 int[]already\_placed = new int[8];  
 int iter =0;  
 for(int i:list)  
 {  
 if(ArrWork.*place*(iter,i,already\_placed)) {  
 already\_placed[iter] = i;  
 iter++;  
 }  
 else break;  
 }  
 return iter == 8;  
 }  
  
  
 static ArrayList<Integer> RBFS(Tree tree, ArrayList<Node>min\_conflicts, ArrayList<Tree> trees\_in\_mem, int iter, Tree primary\_tree, int counter, ArrayList<Integer> answer){  
 if(answer!=null)  
 return answer;  
 if(counter<1000)  
 {  
 if (trees\_in\_mem.size()>1&&counter%100==0)  
 {  
 Tree new\_tree = trees\_in\_mem.get(0);  
 trees\_in\_mem = new ArrayList<>();  
 trees\_in\_mem.add(new\_tree);  
 Node min = min\_conflicts.get(0);  
 min\_conflicts= new ArrayList<>();  
 min\_conflicts.add(min);  
 answer = *RBFS*(new\_tree,min\_conflicts,trees\_in\_mem,iter=1, primary\_tree,counter++,answer);  
 if(answer!=null)  
 return answer;  
 }  
 if(counter%200==0)  
 {  
 trees\_in\_mem = new ArrayList<>();  
 min\_conflicts = new ArrayList<>();  
 answer = *RBFS*(primary\_tree,min\_conflicts,trees\_in\_mem,iter =0,primary\_tree,counter+=1,answer);  
 if(answer!=null)  
 return answer;  
 }  
 Node start = *choose\_less\_conflict\_in\_list*(tree.getRoot().getChildren());  
 if(start!=null && start.amount\_of\_conflicts==0)  
 {  
 System.*out*.println("FOUND");  
  
 return answer= start.get\_col\_list();  
 }  
 else{  
 if(start!=null &&(start.getDepth()==0 || start.getConflicts()<= *find\_min\_node*(min\_conflicts).getConflicts()) ) {  
 Tree new\_tree = new Tree(start);  
 trees\_in\_mem.add(new\_tree);  
 min\_conflicts.add(start);  
  
 answer = *RBFS*(new\_tree,min\_conflicts,trees\_in\_mem,iter+=1,primary\_tree,counter+=1,answer);  
 if(answer!=null)  
 return answer;  
 }  
 else if(trees\_in\_mem.size()>1){  
 trees\_in\_mem.remove(trees\_in\_mem.size()-1);  
 min\_conflicts.remove(min\_conflicts.size()-1);  
 Tree new\_tree = trees\_in\_mem.get(trees\_in\_mem.size()-1);  
  
 answer = *RBFS*(new\_tree, min\_conflicts,trees\_in\_mem,iter-=1,primary\_tree,counter+=1,answer);  
 if(answer!=null)  
 return answer;  
 }  
 else {  
 counter =200;  
 answer = *RBFS*(null, min\_conflicts,trees\_in\_mem,iter-=1,primary\_tree,counter,answer);  
 }  
 }  
 if(answer!=null)  
 return answer;  
 }  
  
  
 return answer;  
 }  
  
  
 static Node find\_min\_node(ArrayList<Node> nodes)  
 {  
 int min = 10000;  
 Node f\_ans = null;  
 for(Node n: nodes)  
 {  
 if(n.getConflicts()<min)  
 {  
 min = n.getConflicts();  
 f\_ans=n;  
  
 }  
 }  
 if(f\_ans!=null) f\_ans.setVisited(true);  
 return f\_ans;  
 }  
  
  
 static Node choose\_less\_conflict\_in\_list(ArrayList<Node> nodes)  
 {  
 int min = 10000;  
 Node f\_ans = null;  
 for(Node n: nodes)  
 {  
 if(n.getConflicts()<min&&!n.visited)  
 {  
 min = n.getConflicts();  
 f\_ans=n;  
  
 }  
 }  
 if(f\_ans!=null) f\_ans.setVisited(true);  
 return f\_ans;  
 }  
  
}

ArrWork.java:

package ArrWork;  
  
import java.util.\*;  
  
  
public class ArrWork {  
  
 public static String[][] create\_matr(int[] arr, int n)  
 {  
 String[][] matr = new String[n][n];  
 for(int i =0; i <n;i++)  
 {  
 for (int j = 0; j <n; j++)  
 {  
 if(arr[i]==j)  
 System.*out*.print(" Q ");  
 else{  
 System.*out*.print(" # ");  
 }  
  
 }  
 System.*out*.println();  
 }  
 return matr;  
 }  
  
  
 public static boolean place(int queen\_number, int column\_number, int[] columns\_placed)  
 {  
 for(int j =0; j<queen\_number;j++)  
 {  
 if(columns\_placed[j]==column\_number || Math.*abs*(columns\_placed[j]-column\_number)==Math.*abs*(j-queen\_number))  
  
 {  
 return false;  
 }  
 }  
 return true;  
 }  
  
 public static ArrayList<Integer> array\_to\_list(int []arr)  
 {  
 ArrayList<Integer> answer = new ArrayList<>();  
 for (int j : arr) {  
 answer.add(j);  
 }  
  
 return answer;  
 }  
  
  
  
 public static int[] make\_rand\_int\_arr (int n)  
 {  
 Random random = new Random();  
 int[]result = new int[n];  
 for(int i = 0; i <n; i++)  
 result[i]=random.nextInt(n);  
 return result;  
 }  
  
 public static int[] arrlist\_to\_arr(ArrayList<Integer>arrlist)  
 {  
 int[] answer = new int[arrlist.size()];  
 int iter =0;  
 for(int i: arrlist) {  
 answer[iter] = i;  
 iter++;  
 }  
 return answer;  
  
 }  
  
 static int amount\_of\_duplicates\_in\_arrlist(ArrayList<Integer> arr, int depth)  
 {  
 ArrayList<Integer> checked = new ArrayList<>();  
 ArrayList<Integer>duplicates = new ArrayList<>();  
 for(int i =0; i<depth;i++)  
 {  
 int a = arr.get(i);  
 if(!checked.contains(a))  
 checked.add(a);  
 else  
 duplicates.add(a);  
 }  
 return duplicates.size();  
 }  
  
}

Tree.java:

package ArrWork;  
import java.util.ArrayList;  
  
public class Tree{  
 private final Node root;  
 public Node getRoot(){  
 return root;  
 }  
 Tree(Node root){  
 this.root = root;  
 }  
 public void generateBranch(Node parent) {  
 int starter = change\_value(parent.get\_col\_list(),0);  
 make\_node\_for\_tree(parent,starter,0);  
 }  
 public void make\_tree(Tree tree)  
 {  
 tree.generateBranch(tree.root);  
  
  
 }  
  
 public Node make\_node\_for\_tree(Node chilld, int starter, int depth)  
 {  
  
 Node chiiild = null;  
 for (int j = 0; j < 8; j++) {  
 ArrayList<Integer> new\_new\_list = new ArrayList<>(chilld.get\_col\_list());  
 new\_new\_list.set(depth, starter);  
 chiiild = new Node(new\_new\_list, depth);  
 starter = change\_value(new\_new\_list,depth);  
 if(depth<7) make\_node\_for\_tree(chiiild,starter,depth+1);  
 chilld.addChild(chiiild);  
 starter = change\_value(new\_new\_list, depth);  
 }  
 return chiiild;  
 }  
  
 int change\_value(ArrayList<Integer>arrayList, int index)  
 {  
 int val = arrayList.get(index);  
 if(val+1>7)  
 val=0;  
 else val++;  
 return val;  
 }  
}

Node.java

package ArrWork;  
  
import java.util.ArrayList;  
  
public class Node {  
 private final ArrayList<Integer> col\_list;  
 private final int depth;  
 private final ArrayList<Node> children ;  
 int amount\_of\_conflicts;  
  
 boolean visited = false;  
  
 Node(ArrayList<Integer> col\_list, int depth){  
 this.col\_list = col\_list;  
 this.children = new ArrayList<>();  
 this.depth = depth;  
 this.amount\_of\_conflicts = F2.*F2*(ArrWork.*arrlist\_to\_arr*(col\_list),8)+ArrWork.*amount\_of\_duplicates\_in\_arrlist*(col\_list,depth)\*10;  
 }  
  
 public int getConflicts ()  
 {  
 return this.amount\_of\_conflicts;  
 }  
 public void setVisited(boolean val)  
 {  
 visited = val;  
 }  
 public ArrayList<Integer> get\_col\_list(){  
 return col\_list;  
 }  
  
  
  
 public void addChild(Node child){  
 children.add(child);  
 }  
 public ArrayList<Node> getChildren() {  
 return children;  
 }  
  
 public int getDepth() {  
 return depth;  
 }  
}

F2.java:

package ArrWork;  
  
public class F2 {  
 public static int F2(int[]array, int n)  
 {  
 int pairs=0;  
 for (int i=0; i<n;i++)  
 {  
 for(int j =i+1; j<n;j++)  
 if(j!=i && Math.*abs*(array[i]-array[j])==Math.*abs*(i-j))  
 pairs++;  
  
 }  
  
 for(int i =0; i<n;i++)  
 {  
 int tmp =0;  
 for(int j =i+1; j<n; j++)  
 if(array[i]==array[j]&& i!=j)  
 tmp++;  
 if (tmp>1){  
 pairs+=tmp;  
 }  
 if(tmp==1)  
 pairs++;  
 }  
  
  
 return pairs;  
 }  
}

### Приклади роботи

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

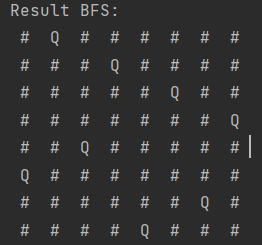
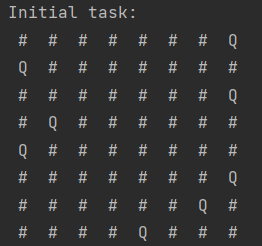


Рисунок 3.1 – Алгоритм BFS

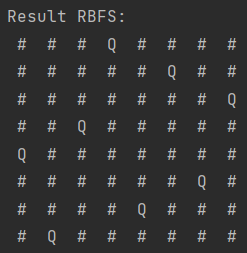
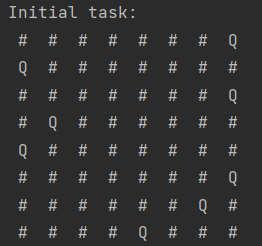


Рисунок 3.2 – Алгоритм RBFS

## Дослідження алгоритмів

В таблиці 3.1 наведені характеристики оцінювання алгоритму BFS, задачі 8 queens для 10 початкових станів.

Таблиця 3.1 – Характеристики оцінювання алгоритму BFS

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Початкові стани | Ітерації | К-сть гл. кутів | Всього станів | Всього станів у пам’яті |
|  | 4 |  | 32768 | 32768 |
|  | 5 |  | 262144 | 262144 |
|  | 5 |  | 262144 | 262144 |
|  | 4 |  | 32768 | 32768 |
|  | 4 |  | 32768 | 32768 |
|  | 3 |  | 4096 | 4096 |
|  | 1 |  | 56 | 56 |

В таблиці 3.2 наведені характеристики оцінювання алгоритму RBFS, задачі 8 queens для 10 початкових станів.

Таблиця 3.3 – Характеристики оцінювання алгоритму RBFS

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Початкові стани | Ітерації | К-сть гл. кутів | Всього станів | Всього станів у пам’яті |
|  | 154 |  | 249 | 6\*56 |
|  | 77 |  | 118 | 6 |
|  | 227 |  | 359 | 7 |
|  | 535 |  | 843 | 7 |
|  | 617 |  | 978 | 7 |
|  | 14 |  | 24 | 7 |
|  | 428 |  | 681 | 5 |

Висновок

* При виконанні даної лабораторної роботи було розглянуто та досліджено алгоритми неінформативного, інформативного та локального пошуку. В часності BFS та RBFS для логічної задачі 8 ферзів. Задача полягає у розташуванні 8 ферзів на полі так, щоб ні один не бив іншого. Проведено порівняльний аналіз ефективності використання алгоритмів. Обидва алгоритми показали правильні варіанти розв’язання задачі, перевірені як вручну, так і використовуючи функцію F2. F2 – кількість пар ферзів, які б’ють один одного без урахування видимості. Усі алгоритми указані в звіті.

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%.