НТУУ «Київський політехнічний інститут імені І. Сікорського»
Факультет інформатики та обчислювальної техніки
Кафедра автоматизованих систем обробки інформації та управління

Лабораторна робота №1
з дисципліни
«Основи штучного інтелекту»

Виконав
студент групи ІС-71
Янголь В. Є.

Перевірила
ст. вик. каф. АСОІУ
Мажара О. О.

Київ
2020

Тема: Методи неінформативного пошуку

Мета: Ознайомитися з алгоритмами неінформативного пошуку та їх властивостями

1. Постановка задачі

Розробити алгоритм вирішення старовинної логічної задачі: «Як за допомогою 5-ти літрового і 9-ти літрового відра набрати з річки 3 літри води?»

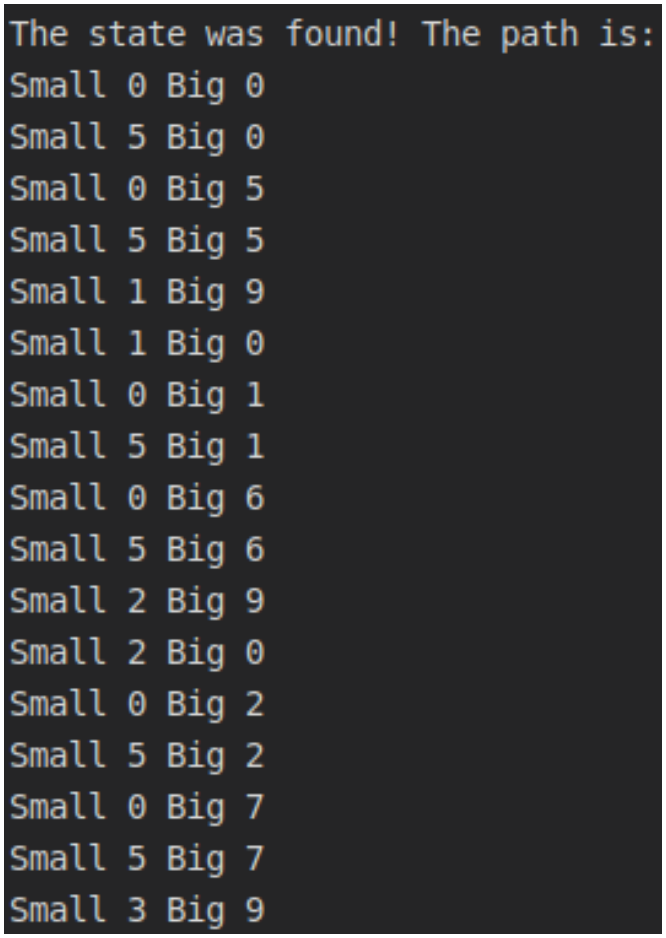
Використати метод пошуку в глибину.

2. Лістинг

Програмний код представлений у Додатку 1 до звіту.

3. Результати роботи програмного забезпечення

Скріншот роботи програми за цільового стану, поданого в умові задачі:



Скріншот роботи програми за тестового цільового стану «набрати 2 літри»:

```
The state was found! The path is:
Small 0 Big 0
Small 5 Big 0
Small 0 Big 5
Small 5 Big 5
Small 1 Big 9
Small 1 Big 0
Small 0 Big 1
Small 5 Big 1
Small 0 Big 6
Small 5 Big 6
Small 2 Big 9
```

4. Висновок

Я був ознайомлений з методами неінформативного пошуку в межах заданого простору станів та з властивостями цих методів, а також реалізував неінформативний пошук у глибину.

Пошук у глибину заснований на рекурсії: якщо деякий стан не є цільовим, то спочатку переглядаються його нащадки.

У графічному представленні (на прикладі обходу дерева) спочатку алгоритм перебирає найбільш «ліві» стани, і якщо серед них немає цільового, то переходить до тих, що знаходяться «правіше».

Пошук в глибину гарантує повернення розв’язку, якщо він існує (тобто алгоритм є повним), але не гарантує повернення найкоротшого шляху, оскільки повертає першу вершину, що відповідає цільовому стану (тобто алгоритм не є оптимальним).

5. Відповіді на контрольні запитання

1. Пошук в глибину на дереві станів <V, E>:

$$
\begin{aligned}
&\textbf{function } DepthFirstSearch(V, E, Goal) \rightarrow a\ solution\ or\ failure \\
&\quad \textbf{foreach } node \textbf{ in } V \\
&\quad\quad node.visited := false \\
&\quad currentNode := V_0 \\
&\quad result := ReviewNode(currentNode) \\
&\quad \textbf{return } result
\end{aligned}
$$

```
function ReviewNode(V_i, Goal) → a solution or failure or cutoff
    if V_i.visited
        then return cutoff
    V_i.visited := true
    if V_i = Goal
        then return V_i.path
    nextNodes := V_i.Expand()
    foreach node in nextNodes
        result := ReviewNode(node, Goal)
        if result = Goal
            then return result
    return failure
```

Пошук в ширину на дереві станів <V, E>:

```
function BreadthFirstSearch(V, E, Goal) → a solution or failure
    foreach node in V
        node.visited := false
    firstNode := V_0
    queue := empty queue
    queue.Enqueue(firstNode)
    while queue.Count ≠ 0
        currentNode := queue.Dequeue()
        if currentNode.visited
            then continue
        currentNode.visited := true
        if currentNode = Goal
            then return currentNode.result
        nextNodes := currentNode.Expand()
        foreach node in nextNodes
            queue.Enqueue(node)
    return failure
```

2. Ідея пошуку з обмеженням глибини: відсікаються ті стани, що лежать нижче заданого рівня. Таким чином, якщо існує розв’язок, що містить меншу кількість переходів, ніж заданий рівень, то він буде повернений замість потенційних розв’язків, що знаходяться на нижчих рівнях та що могли б бути розглянуті раніше у класичному алгоритмі пошуку в глибину.

3. Алгоритми пошуку в глибину, пошуку в ширину, пошуку з ітеративним заглибленням та пошуку з рівномірною вартістю є повними (гарантують повернення розв’язку, якщо він існує). Алгоритм пошуку з ітеративним заглибленням не є повним, оскільки якщо оптимальний шлях до вершини має більшу довжину, ніж вказана кількість рівнів, то цей шлях не буде розглянутий.

Оптимальним за будь-яких умов є пошук з рівномірною вартістю, оскільки він розглядає шляхи не за кількістю переходів, а за вагою ребер, за якими може відбуватися перехід. За умови, якщо ваги всіх ребер однакові (тобто у випадку, якщо пошук за вагами ребер та пошук за кількістю переходів еквівалентні), оптимальними також є алгоритми пошуку в глибину та пошуку з ітеративним заглибленням.

Додаток 1

Лістинг

Файл State.cs (опис стану):

```
using System;
using System.Collections.Generic;

namespace Yanhol1
{
    public class State
    {
        public int small; // amount of water in small container
        public int big; // amount of water in big container

        public List<string> path;
        public State previousState;

        public State()
        {
            small = 0;
            big = 0;
            path = new List<string>();
            previousState = null;
        }

        public State(State previous, int newSmall, int newBig)
        {
            small = newSmall;
            big = newBig;

            path = new List<string>(previous.path);
            path.Add("Small " + newSmall.ToString() + " Big " +
newBig.ToString());
            previousState = previous;
        }

        // Returns true if states have identical configuration of elements
        public bool StatesAreEqual(State compared)
        {
            if (this == null || compared == null)
                return false;
            if (small == compared.small && big == compared.big)
                return true;
            return false;
        }
    }
}
```

Файл Program.cs:

```
using System;
using System.Collections.Generic;

namespace Yanhol1
{
    internal class Program
    {
        private static HashSet<State> reviewedStates;

        private static bool IsGoalState(State currentState)
        {
            if (currentState == null)
                return false;

            if (currentState.small == 3 || currentState.big == 3 ||
currentState.small + currentState.big == 3)
                return true;

            return false;
        }

        // Returns true if there's cycle in the path
        private static bool IsCycle(State node)
        {
            State current = node.previousState;
            while (true)
            {
                if (current == null)
                    return false;
                else if (current.StatesAreEqual(node))
                    return true;

                current = current.previousState;
            }
        }

        // Returns all the possible moves from the parametre node
        private static List<State> GetPossibleMoves(State previousState)
        {
            List<State> possibleMoves = new List<State>();

            State smallContainerToMax = new State(previousState, 5,
previousState.big);
            if (!previousState.StatesAreEqual(smallContainerToMax)
                && smallContainerToMax.small >= 0 && smallContainerToMax.small
<= 5
                && smallContainerToMax.big >=0 && smallContainerToMax.big <= 9)
                possibleMoves.Add(smallContainerToMax);

            State bigContainerToMax = new State(previousState,
previousState.small, 9);
            if (!previousState.StatesAreEqual(bigContainerToMax)
                && bigContainerToMax.small >= 0 && bigContainerToMax.small <= 5
                && bigContainerToMax.big >=0 && bigContainerToMax.big <= 9)
                possibleMoves.Add(bigContainerToMax);

            State smallContainerToMin = new State(previousState, 0,
previousState.big);
            if (!previousState.StatesAreEqual(smallContainerToMin)
```

```
                && smallContainerToMin.small >= 0 && smallContainerToMin.small
<= 5
                && smallContainerToMin.big >=0 && smallContainerToMin.big <= 9)
                possibleMoves.Add(smallContainerToMin);

            State bigContainerToMin = new State(previousState,
previousState.small, 0);
            if (!previousState.StatesAreEqual(bigContainerToMin)
                && bigContainerToMin.small >= 0 && bigContainerToMin.small <= 5
                && bigContainerToMin.big >=0 && bigContainerToMin.big <= 9)
                possibleMoves.Add(bigContainerToMin);

            // pouring all the water from small to big:
            int emptyInBig = 9 - previousState.big;
            int toBeFilledFromSmall = (previousState.small > emptyInBig) ?
emptyInBig : previousState.small;
            State fillingFromSmall = new State(previousState,
previousState.small - toBeFilledFromSmall,
                previousState.big + toBeFilledFromSmall);
            if (!previousState.StatesAreEqual(fillingFromSmall)
                && fillingFromSmall.small >= 0 && fillingFromSmall.small <= 5
                && fillingFromSmall.big >=0 && fillingFromSmall.big <= 9)
                possibleMoves.Add(fillingFromSmall);

            // pouring all the water from small to big:
            int emptyInSmall = 5 - previousState.big;
            int toBeFilledFromBig = (previousState.big > emptyInSmall) ?
emptyInSmall : previousState.big;
            State fillingFromBig = new State(previousState, previousState.small
+ toBeFilledFromBig,
                    previousState.big - toBeFilledFromBig);
            if (!previousState.StatesAreEqual(fillingFromBig)
                && fillingFromBig.small >= 0 && fillingFromBig.small <= 5
                && fillingFromBig.big >=0 && fillingFromBig.big <= 9)
                possibleMoves.Add(fillingFromBig);

            return possibleMoves;
        }

        // Depth first search - Returns the final state
        public static State DepthFirstSearch(State currentState)
        {
            foreach (var state in reviewedStates)
                if (currentState.StatesAreEqual(state))
                    return null; // cutoff
            reviewedStates.Add(currentState);

            if (IsGoalState(currentState))
                return currentState;

            List<State> possibleStates = GetPossibleMoves(currentState);
            foreach (var state in possibleStates)
            {
                State branchResult = DepthFirstSearch(state);
                if (IsGoalState(branchResult))
                    return branchResult;
            }

            return null;
        }

        public static void Main(string[] args)
        {
            // Solution solution = new Solution();
```

```
            State initialState = new State();
            reviewedStates = new HashSet<State>();
            State finalState = DepthFirstSearch(initialState);
            Console.WriteLine("The state was found! The path is:");
            Console.WriteLine("Small {0} Big {1}", initialState.small,
initialState.big);
            for (var i = 0; i < finalState.path.Count; i++)
                Console.WriteLine(finalState.path[i]);
        }
    }
}
```