Problem Solving Search Algorithm

(Documentation)

Brief explanation about 8-tileslide puzzle problem

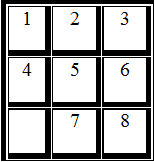
The 8-puzzle is a square board with 9positions, filled by 8 numbered tiles and one gap. At any point, a gap can be swapped with an adjacent tile, creating a new gap position.

Goal is to arrange each tile so it's in the below order

|  |  |  |
| --- | --- | --- |
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 |  |

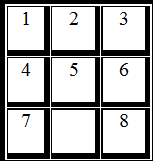
For example:

If initial state is "123456078" , tiles should be swapped until we reach goal state "123456780"



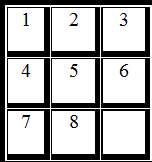
0 is representated as blank space.

Initial state: 123456078



7 is swapped with 0

Intermediate State: 123456708



8 is swapped with 0

Goal State: 123456708

Solving tileslide puzzle problem

I implemented using Iterative Deepening Search method and A\* method.

1. Iterative deepening search method.

It works same as DFS but here we will pass depth(incremental wise) that is up to which level it should traverse

Pseudo code.

Initial\_State = "123456078";

Goal\_State = "123456780"

Goal\_found = 0;

level = 0;

for(i=0; i<=Level && Goal\_found == 0 ;i++)

{

Int Depth = i;

Depth\_Limited\_Search(Initial\_State, Goal, Depth);

Level++;

}

Depth\_Limited\_Search(Int[][] State, int[][] Goal, int Depth)

{

Traverse

if Goal Found then

Goal\_found = 1;

}

Output:

Program Output:

\*\*\*\*\*\*\*\*\*\*\*\*\*Tile\_Slide Problem - Iterative Deepening\*\*\*\*\*\*\*\*\*\*\*\*\*

1. 3\*3

2. 4\*4

Enter your option

1

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*(3\*3)\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Please enter the Initial State as a string

123456078

Please enter the Goal State as a string

123456780

Initial State

1 2 3

4 5 6

0 7 8

Next Action : R

1 2 3

4 5 6

7 0 8

Next Action : R

1 2 3

4 5 6

7 8 0

Sequence of Action: RR

Goal State

1 2 3

4 5 6

7 8 0

2. A\* method

The central idea in the so-called A\* algorithm is to guide best-first search both by the estimate to the goal as given by the heuristic function h and the cost of the path developed so far.

Let n be a node, g(n) the cost of moving from the initial node to n along the current path, and h(n) the estimated cost of reaching a goal node from n.

Define f(n) as follows: f(n) = g(n) + h(n) This is the estimated cost of the cheapest path through n leading from the initial node to a goal node. A\* is the best-first search algorithm that always expands a node n such that f(n) is minimal

Pseudo code:

Find\_Manhattan\_Distance(Object Obj,Object Goal\_Obj)

{

Calculate Manhattan count

**return** Manhattan\_Count;

}

**public** **static** **int** Create\_Child(Astar[] Array,Astar Parent,Astar Goal,**int** NextIndex)//Creation of child nodes

{

**int** Index=NextIndex;

**if**(*Up\_Direction*(Parent))

{

Array[Index]=**new** Astar(Parent.State,Parent.Zero\_X\_pos,Parent.Zero\_Y\_pos,Index,Parent.Address,'U');

*Swap\_Up*(Array[Index]);

Array[Index].Zero\_X\_pos--;

Array[Index].ManhattanDistance=*Find\_Manhattan\_Distance*(Array[Index],Goal);

**boolean** cond=*Compare\_Node*(Array[Index],Array[Parent.Parent\_address]);

**if**(cond==**false**)

{

Array[Index]=**null**;

}

**else**

{

*PrintAstar*(Array[Index]);

++Index;

}

}

**if**(*Down\_DirectionDirection*(Parent))

{

Array[Index]=**new** Astar(Parent.State,Parent.Zero\_X\_pos,Parent.Zero\_Y\_pos,Index,Parent.Address,'D');

*Swap\_Down*(Array[Index]);

Array[Index].Zero\_X\_pos++;

Array[Index].ManhattanDistance=*Find\_Manhattan\_Distance*(Array[Index],Goal);

**boolean** cond=*Compare\_Node*(Array[Index],Array[Parent.Parent\_address]);

**if**(cond==**false**)

{

Array[Index]=**null**;

}

**else**

{

*PrintAstar*(Array[Index]);

++Index;

}

}

**if**(*Right\_Direction*(Parent))

{

Array[Index]=**new** Astar(Parent.State,Parent.Zero\_X\_pos,Parent.Zero\_Y\_pos,Index,Parent.Address,'R');

*Swap\_Right*(Array[Index]);

Array[Index].Zero\_Y\_pos++;

Array[Index].ManhattanDistance=*Find\_Manhattan\_Distance*(Array[Index],Goal);

**boolean** cond=*Compare\_Node*(Array[Index],Array[Parent.Parent\_address]);

**if**(cond==**false**)

{

Array[Index]=**null**;

}

**else**

{

*PrintAstar*(Array[Index]);

++Index;

}

}

**if**(*Left\_Direction*(Parent))

{

Array[Index]=**new** Astar(Parent.State,Parent.Zero\_X\_pos,Parent.Zero\_Y\_pos,Index,Parent.Address,'L');

*Swap\_Left*(Array[Index]);

Array[Index].Zero\_Y\_pos--;

Array[Index].ManhattanDistance=*Find\_Manhattan\_Distance*(Array[Index],Goal);

**boolean** cond=*Compare\_Node*(Array[Index],Array[Parent.Parent\_address]);

**if**(cond==**false**)

{

Array[Index]=**null**;

}

**else**

{

*PrintAstar*(Array[Index]);

++Index;

}

}

**return** Index;

}

**public** **static** **void** main(String[] args)

{

**int** blankXstate=0;

**int** blankYstate=0;

System.***out***.println("A Star implementation for Tile Sliding puzzle-----\n\n");

System.***out***.println("Enter your option"+"\n 1.3\*3 \n 2.4\*4\n");

Scanner scanner = **new** Scanner(System.***in***);

**int** option=scanner.nextInt();

**if**(option==1)

{

Astar[] ObjectArray=**new** Astar[400000];

Astar Goal=**new** Astar();

Goal.State = {1,2,3,4,5,6,7,8,0};

**int** InitialList[][]=**new** **int**[5][5];

InitialList[0][0]=InitialList[0][1]=InitialList[0][2]=InitialList[0][3]=InitialList[0][4]=InitialList[1][0]=InitialList[1][4]=InitialList[2][0]=InitialList[2][4]=InitialList[3][0]=InitialList[3][4]=InitialList[4][0]=InitialList[4][1]=InitialList[4][2]=InitialList[4][3]=InitialList[4][4]=-1;

System.***out***.println("Enter the Initial List as a sequence of integers");

**for**(**int** i=1;i<4;i++)

**for**(**int** j=1;j<4;j++)

InitialList[i][j]=scanner.nextInt();

**for** (**int** i=1; i<4; i++)

{

**for** (**int** j=1; j<4; j++)

{

**if**(InitialList[i][j]==0)

{

blankXstate = i;

blankYstate = j;

}

}

}

**int** count=0;

ObjectArray[0]=**new** Astar(InitialList,blankXstate,blankYstate,0,0,'\0');

ObjectArray[0].ManhattanDistance=*Find\_Manhattan\_Distance*(ObjectArray[0],Goal);

System.***out***.println("The Root Node is:");

*PrintAstar*(ObjectArray[0]);

System.***out***.println("The New Child nodes created are:"+"\n\n");

**int** Next\_Node\_Index=1;

**int** i=0;

**for**(i=0;*Compare\_Node*(ObjectArray[i],Goal); i++)

{

Next\_Node\_Index=*Create\_Child*(ObjectArray,ObjectArray[i],Goal,Next\_Node\_Index);

}

count=i;

System.***out***.println("\nThe solution path is:");

*PrintAstar*(ObjectArray[count]);

*ApplyAstar*(count,ObjectArray);

}

**else** **if**(option == 2)

{

Node\_15\_puzzle[] ObjectArray=**new** Node\_15\_puzzle[300000];

Node\_15\_puzzle Goal=**new** Node\_15\_puzzle();

//initialize the goal state matrix

int[][] Goal\_state = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,0};

System.***out***.println("Enter the Initial List as a sequence of integers");

**for**(**int** i=1;i<5;i++)

**for**(**int** j=1;j<5;j++)

InitialList[i][j]=scanner.nextInt();

**for** (**int** i=1; i<5; i++)

{

**for** (**int** j=1; j<5; j++)

{

**if**(InitialList[i][j]==0)

{

blankXstate = i;

blankYstate = j;

}

}

}

**int** count=0;

ObjectArray[0]=**new** Node\_15\_puzzle(InitialList,blankXstate,blankYstate,0,0,'\0');

ObjectArray[0].ManhattanDistance=*Find\_Manhattan\_Distance1*(ObjectArray[0],Goal);

System.***out***.println("The root Astar is:");

*PrintNode\_15\_puzzle*(ObjectArray[0]);

System.***out***.println("The child Astars created are:"+"\n\n");

**int** Next\_Node\_Index=1;

**int** i=0;

**for**(i=0;*Compare\_Node1*(ObjectArray[i],Goal); i++)

{

Next\_Node\_Index=*Create\_Child1*(ObjectArray,ObjectArray[i],Goal,Next\_Node\_Index);

}

count=i;

System.***out***.println("\nThe solution path is:");

*PrintNode\_15\_puzzle*(ObjectArray[count]);

*ApplyAstar1*(count,ObjectArray);

}

**else** **if**(option!= 1 && option!=2)

{ System.***out***.println("Invalid Option Selected");

System.*exit*(0);

}

scanner.close();

}

}