UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

Heuristic function in an algorithm of First-Best search for the problem of Tower of Hanoi: optimal route for n disks

ARTÍCULO ACADÉMICO

Que para obtener el título de

Ingeniero en Computación

P R E S E N T A

Erick Berssaín García Ventura

ASESORA DE ARTÍCULO ACADÉMICO

M. en I. Norma Elva Chávez Rodríguez

Ciudad Universitaria, Cd. Mx., 2018

Heuristic function in an algorithm of First-Best search for the problem of Tower of Hanoi: optimal route for n disks.

Erick Berssaín García V., Norma Elva Chávez R.
School of Engineering, National Autonomous University of México, México City

Abstract - *Along this paper we propose a new algorithm for solving the Tower of Hanoi puzzle. We focus on computer memory efficiency by applying the Artificial Intelligence's method "Best-First Search". We develop the algorithm with detailed explanations, taking in mind it may be coded in any programming language. Furthermore, we set a heuristic function and several mathematical definitions, diagrams, and examples in order to make the reader understand fully. Finally, we present the solution implemented in python, adding proofs of its logical functioning as well as results about execution time made for 16 cases (from using only 3 disks until 18).*

Keywords: *Tower of Hanoi, best-first search, heuristic, function, python.*

1 Introduction

The Tower of Hanoi, mathematical puzzle, is an example to apply programming techniques such as recursive algorithms [1][2]. Recursive algorithms are relatively simple to implement in most programming languages. These algorithms aid to solve a wide variety of problems and have been selected, by software companies such as Microsoft [8], Oracle, and Facebook, and at robotics challenges [3], to test the aspirants programming abilities. Mainly in the first stage of the recruitment process. These companies change the constraints of the puzzle to increase the complexity of the algorithm. For example, changing the number of disks the algorithm has to solve for.

However, as the authors describe in [5], solving this problem by a recursive algorithm takes up a lot of computing resources. And, although many algorithms have been developed, few times they are measured in efficiency and memory spending.

Thereby we put forward a method where we minimize the use of memory.

First we take as a base a 3D model from the original Hanoi Tower. From there, we make a geometric projection to a 2D model (new model). Then, we redefine the allowed movements from 3D model, now for the 2D model. At the end we abstract the new model like an array or a list for each tower (depending on the programming langue which is talking on).

Having this abstraction, we proceed to establish an evaluation function which is the main point of the algorithm.

Solving this problem by different methods has a great relevance. It is not only related with the puzzle/game, but also with planning and expanding to other areas. Graphs Theory [6], Ad Hoc Networks [7], and Collaborative/Educative [4] issues have been proposed from this Tower of Hanoi problem.

2 Considerations

Before to start the algorithm explanation, next are some concepts that are used to help the understanding of the whole paper.

Consideration 1: We refer to "n", which means the total number of disks selected to play, and whose definition is the follows:

$$n = \{\eta \mid \eta \epsilon \mathbb{N}\} \qquad (1)$$

Consideration 2: Although most of the examples in this paper are showed for 3 or 4 disks, the solution applies for n number of disks.

Consideration 3: The algorithm and solution proposed has been tested for 3 pegs.

Consideration 4: Since the puzzle may have n disks, we refer to disks weights according to their size. The smallest one has a weight of 1 and the largest one has n. As shown in figure 1.

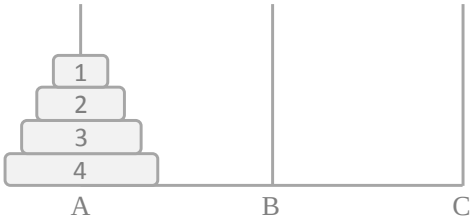

Figure 1
Weight assignments to each disk

Consideration 5: The optimum number of movements for solving the puzzle, with n disks is:

$$M_n = 2^n\text{-}1 \qquad (2)$$

*This is a puzzle fact and it has already demonstrated and taken as a true in other papers [1].

Consideration 6: Figure 3 is the flat (2D) representation of the 3D puzzle model (Figure 2).

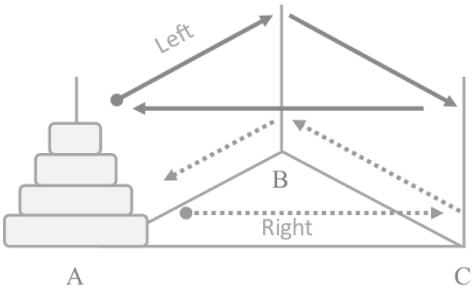

Figure 2
3D Representation of Tower of Hanoi

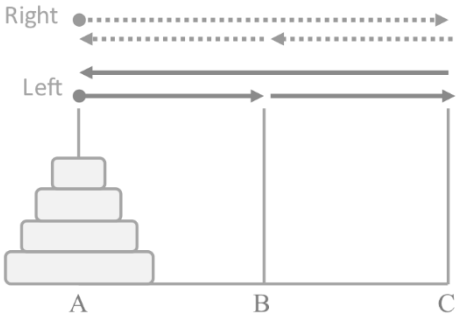

Figure 3
2D Representation of Tower of Hanoi

For the movements to the right, they are in the next sequence:

Origin Tower →Destination Tower

A→C

C→B

B→A

Meanwhile for the ones to the left, they are:

A→B

B→C

C→A

3 Premises

With the aim of getting the problem solved with the minimum of movements, it was firstly studied without the programming perspective. Next we define some premises which are the sustention of the algorithm.

3.1 Premise 1: Definition

Premise 1 - Direction of movement.

If the number of disks which are played is an even number, then:

- *The disks with even weight (2, 4, 6, etc.) have to be moved to the left.*
- *The disks with odd weight (1, 3, 5, etc.) have to be moved move to the right.*

Otherwise, if the number of played disks is an odd number the directions are exchanged.
Thus:

- *Even weight disks to the right.*
- *Odd weight disks to the left.*

3.2 Premise 1: Examples

For both of next examples, disks filled with dots move to the right and disks filled with lines to the left.

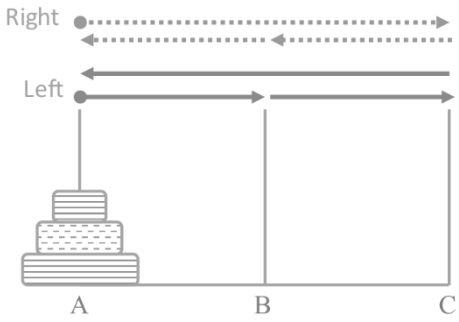

Figure 4
When the number of disk played is odd.

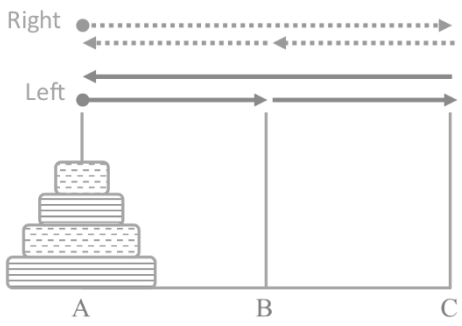

Figure 4
When the number of disk played is even.

3.3 Premise 2: Definition

Premise 2 - Number of movement.

For all disks in puzzle, disk number "k" whose weight is represented with W_k and where $W_k = k$, is only allowed to be moved every 2^{W_k} times. Counting from the first time it was moved, which is the $2^{W_k - 1}$ time.

3.4 Premise 2: Examples

Disk 1

$$k = W_k = 1$$

First movement at $2^{1-1} = 1$
Move every $2^1 = 2$ times.
Expected to move at time: 1, 3, 5…

Disk 2

$$k = W_k = 2$$

First movement at $2^{2-1} = 2$
Move every $2^2 = 4$ times.
Expected to move at time: 2, 6, 10 …

Disk 3

$$k = W_k = 3$$

First movement at $2^{3-1} = 4$
Move every $2^3 = 8$ times.
Expected to move at: 4, 12, 20 …

And so on for all disks.

4 Heuristic function

As in the title is set, the method we developed to find the best route is through a First-Best Search. There we evaluate the 2 parameters raised in the premises from Section 3 on this paper.

Thereby, it is possible to make a heuristic evaluation function called h(k), which in turn is made up of two other functions, d(k) and t(k).

*Note: Here, we take k as it was used in section 3.3.

Where $d(k)$ evaluates ***Premise 1***, and it is defined as:

$$d(k) = \begin{cases} 1, & \text{if k can be moved to its expected direction.} \\ & \text{or} \\ 3, & \text{if k cannot be moved to its expected direction.} \end{cases} \quad (3)$$

"d" of Direction

And t(k) evaluates ***Premise 2***, and it is defined as:

$$t(k) = \begin{cases} 1, & if\ \ it\ is\ time\ to\ move\ k \\ & or \\ 3, & if\ it\ is\ not\ time\ to\ move\ k \end{cases} \quad (4)$$

"t" of Time

By adding both of above functions, we have the main function:

$$h(k) = d(k) + t(k) \quad (5)$$

"h" of Heuristic

So, there may be 3 results: 2, 4, and 6. Thus, the algorithm provides us certainty that there is only one possible disk to move for each step. This one will be chosen by taking as criterion the **smallest value of heuristics**.

$$h_{best}(k) = d_{best}(k) + t_{best}(k) \quad (6)$$

With the criterion of smallest value of heuristics, joined with definitions of d(k) and t(k).

$$t_{best}(k) = 1$$

and

$$d_{best}(k) = 1$$

This means the algorithm will always choose the movement whose $h(k) = h_{best}(k) = 2$.

5 Design of solution

The algorithm developed has been designed to be coded by a functional structure. It has 7 functions. 6 of them are for auxiliary procedures by the main thread. The other one is an additional function which is used to verify the correct functioning of the program and to observe what happens at each step.

The general purpose of each function is described below.

5.1 Make times

Arguments: Number of disks for playing.

First it creates the initial state for each tower. Always tower A is the initial tower (the one which has all disks). And also creates our goal state, always tower C is the target destination.

For instance, for 3 disks our initial state would be:

$$\begin{aligned} \text{TA} &= [3,2,1] \\ \text{TB} &= [\ \ ] \\ \text{TC} &= [\ \ ] \end{aligned}$$

and the goal state:

$$\begin{aligned} \text{TA} &= [\ \ ] \\ \text{TB} &= [\ \ ] \\ \text{TC} &= [3,2,1] \end{aligned}$$

Then, also it makes as many lists as number of disks. Each list is assigned to a particular disk. There we save the expected times for moving that disk, premise 2.

Taking the examples from premise 2, we have for each disk the next lists:

$$\begin{aligned} disk\ 1 &= [1, 3, 5, 7 \dots] \\ disk\ 2 &= [2, 6, 10, 14 \dots] \\ disk\ 3 &= [4, 12, 20, 28 \dots] \end{aligned}$$

Knowing the algorithm executes just the needed number of steps, we restrict the amount of elements per list. We let the lists grow up while their elements are smaller than $2^n - 1$ (from consideration 4, minimum number of needed steps).

This way, lists are defined as:

$$\begin{gathered} L_k : List\ of\ disk\ "k" \\ |L_k| : Size\ (number\ of\ elements)\ of\ L_k \\ With\ \ k \in \mathbb{N}\ , k \leq n \\ L_k = [a_1, a_2, a_3, \dots, a_{i-1}, a_i] \mid a_i > a_{i-1}\ , a_i < 2^n - 1 \\ |L_k| = \frac{2^n}{2^k} \end{gathered} \tag{7}$$

And so, we avoid a memory unnecessary expense.

Here, we have a relation between the weight of the disk and the amount of elements in its list. The smallest disk will be the one which has more elements, while the biggest one will ever have just a single element.

5.2 Last disk value

Arguments: A tower.

It returns the weight of its last disk.

5.3 Expected direction

Arguments: The weight of a single disk.

According to the number of played disks and the disk's weight received, it evaluates Premise 2. Returns the corresponding direction to move the disk.

5.4 Check movement

Arguments: Two towers (origin and target).

It checks if it is possible to move the origin's last disk to target tower. It considers the puzzle rule: *"A disk cannot be over other disk whose weight is less"*.

Here may be 4 possible scenarios to face:

1. Since we receive just the towers, we do not know if the origin tower really has disks in it. If the origin tower is empty, the function returns that the movement is not possible (because it does not have a disk to move).
2. If the target tower is empty, any disk can be moved there. So, the function returns that the movement is possible.
3. If the last disk of the target tower has a less weight than the last disk from origin tower. The function returns that the movement is possible.
4. The opposite case of the previous one. When the last disk of the target tower has a greater weight than the last disk from origin tower. The function returns that the movement is not possible

5.5 Move dish

Arguments: Two towers, origin and target.

It moves origin's last disk to target tower.

5.6 Simulate move dish **Additional function*

Arguments: Two towers (origin and target), a heuristic value.

In order to get evidence about the procedure was followed correctly, we print on a file the all possible states for each step.

5.7 Main

Below we show the program logic and flow, by using previously described functions.

- START
- Ask the number of disks for playing.
- Execute *Make Times*.
- While we do not achieve our goal state:
    - For each Tower, checks if its last disk can be moved to one or both of the other two towers with *Check Movement.*
        - If the movement can be executed:
            - Saves a row from the movement with next values: origin tower and direction of displacement. Besides we add an empty field for heuristic value initialized with zero.
    - For each saved row in previous cycle:
        - Checks if the saved direction on the row is the one expected, according to the lists created by *MakeTimes* (premise 1).
            - If it is: Add 1 to the heuristic value.
            - If not: Add 3 to the heuristic value.
        - Checks if it is time to move last disk for tower in row (premise 2).
            - If it is: Add 1 to the heuristic value.
            - If not: Add 3 to the heuristic value.
    - Select the row with smallest heuristic value.
    - Execute the corresponding movement in the marked direction.
    - Free from memory and all rows saved.
    - Increments in one the number of executed steps.
- FINISH

6 Results

Next we present some results gotten from implementation made in python. Whole evaluation was made under next computer conditions:

- Processor: 2.7 GHz Intel Core i5
- Memory: 8 GB 1867 MHz DDR3
- Storage type: SSD
- OS: macOS Sierra
    - Version 10.12.3
    - Python 2.7

6.1 Procedure

In Figure 6 we show a random part from the file we used for examine the procedure. It is the mentioned on the function Simulate move dish, from previous section.

In this case we played 16 disks. As we can see, we have:

- The sum of executed movements until the moment.
- Current state for the three towers.
- Possible movements that can be made.
- Heuristic values for all those movements.

```
Movement number: 30690
Current state of Towers:
TA: [16, 11, 10, 9, 8, 7, 6, 1]
TB: [15, 14, 13]
TC: [12, 5, 4, 3, 2]

Possible Movements:
                        Heuristics value: : 6
        TA: [16, 11, 10, 9, 8, 7, 6]
        TB: [15, 14, 13]
        TC: [12, 5, 4, 3, 2, 1]

                        Heuristics value: : 4
        TA: [16, 11, 10, 9, 8, 7, 6]
        TB: [15, 14, 13, 1]
        TC: [12, 5, 4, 3, 2]

                        Heuristics value: : 2
        TA: [16, 11, 10, 9, 8, 7, 6, 1]
        TB: [15, 14, 13, 2]
        TC: [12, 5, 4, 3]
Movement number: 30691
Current state of Towers:
TA: [16, 11, 10, 9, 8, 7, 6, 1]
TB: [15, 14, 13, 2]
TC: [12, 5, 4, 3]

Possible Movements:
                        Heuristics value: : 4
        TA: [16, 11, 10, 9, 8, 7, 6]
        TB: [15, 14, 13, 2]
        TC: [12, 5, 4, 3, 1]

                        Heuristics value: : 2
        TA: [16, 11, 10, 9, 8, 7, 6]
        TB: [15, 14, 13, 2, 1]
        TC: [12, 5, 4, 3]

                        Heuristics value: : 6
        TA: [16, 11, 10, 9, 8, 7, 6, 1]
        TB: [15, 14, 13]
        TC: [12, 5, 4, 3, 2]
```

Figure 6
Evaluation example

In state from step 30690, we can observe the movement with the minimum heuristic is the third one. For this movement, disk 2 is moved from tower C (TC) to tower B (TB). We can verify the movement was correctly chosen on step 30691, where we observe the movement completed and the new possible movements for that new state.

6.2 Time

About execution time, we tested the program with only processes interns from computer executing. This means no apps such as office suite software or internet browsers were open.

In addition, it is necessary to mention we modified a little the program for this evaluation. We let only essential

functions by removing comments and printing files functions. All these with the aim of measure effective time of the algorithm.

In next table we have data about execution time. Column called Time shows the truncated mean at 20%. Since we took 10 events per disk, that percentage eliminates the 2 biggest and the 2 smallest values. At the end, those times (in seconds) represents the mean of 6 measurements.

We did this because we wanted to have a representative value of the samples. So we eliminated the 4 most scattered values of the mean.

Another thing to mention is we started evaluations with 3 disks because it is the most basic example at finding the puzzle. From 3 to 16 disk, we took 10 measurements which in total were 140. In addition, because of the increase of the lapse of time starting from 17 disks, we took isolated data for those cases.

Table (1)

Number of disks	Time [seconds]		
3	0.00031364	11	0.08890906
4	0.000542223	12	0.27752012
5	0.001012713	13	0.988660783
6	0.001990587	14	3.697941035
7	0.00352335	15	14.26059395
8	0.005911112	16	56.51282227
9	0.012826025	17 Single value	263.5115719
10	0.030550957	18 Single value	1285.241847

In Figure 7, we see an exponential behavior. It continues with the same growing tendency by incrementing the number of disks. In order to have a better visualization, were only graphed the first 8 data.

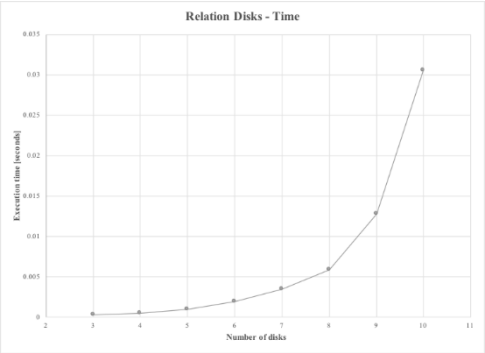

Figure 7
Time tendency

7 Discussion

In [1] authors have already discussed the importance of understanding and manage some concepts we also use in this paper: size of disks, direction of movement (they called cycles), and time to move each disk. Although they justify most of their algorithm's structures, their quantity of propositions, theorems, and properties is kind of large.

Their work is so relevant because sets principles we use. Nevertheless, total of restrictions they have, we summarize in two premises which in turn forms a single heuristic function to evaluate. As well we propose mathematical expressions for some of their propositions.

Besides, we provided an explicit explanation of the logic of functioning. We structured the solution in such way the reader can apply the algorithm in their chosen programming language. By implementing simple functions with generic (not advanced) programming knowledge.

Moreover, the time our algorithm performs is significantly larger than in other papers [1][2]. This situation is not due the malfunctioning or planning of the solution, but the computing requirements. Whereas we save considerably memory because we only keep the last state of towers, in each iteration we have to calculate the heuristic value for the movements. From the file we use to follow the procedure (Figure 6), we noticed there is always 3 possible movements to do in each step (there is an exception on first movement, where there are just 2 possible movements). Thus, we may know how many evaluations we will do:

$$N_e : Total\ number\ of\ \ heuristic\ evaluations \qquad (8)$$

$$N_e = 3 * (2^n - 2) + 2$$

This is a peculiar issue in heuristic algorithms. Faced with not informed searches, where does not really matter optimization, and memory spending is high. Our particular heuristic search takes care about memory and optimization of routes.

8 Concluding and future work

It was set from the very start our solution applies for 3 towers/pegs. Such as they do in [2] where they prove with 4 pegs, one very interesting improvement for our algorithm would be trying to implement it for at least 1 extra tower. We do not discard following looking for upgrades in our code line in order to make performance more efficient.

Finally, we share our code with the aim of challenge readers to incorporate more features and get out a research before of us.

9 Python code

```
# -*- coding: utf-8 -*-
from time import time #import "time" function to test
# Erick Berssain Garcia Ventura
#Developed at School of Engineering, UNAM
#Mexico, Mexico city
#berss4x@hotmail.com , berssain@hotmail.com
# https://berssain.com

generatedStates = open('generatedStates.txt', 'w')
solutionRoute= open('solutionRoute.txt', 'w')
Ti=[[],[],[]]  #List for initial state of each tower (3 towers)
Tg=[[],[],[]]  #List for goal state of each tower (3 towers)
states=[[],[],[]] #states=[[towerOrigin],[last disk's value],[heuristic]]
times=[]
temp=[]
counterTime=1
```

```
#making a list to each disk to know  when its movement shall execute
def makeTimes():
    for x in xrange(0,numberDisks):
        times.extend([[]])
    for x in reversed(xrange(0,numberDisks)):
        temp.append(x+1)
    for x in xrange(0,numberDisks):
        Ti[0].append(temp[x]) #sets the initial state according the number of disks EG: [[4,3,2,1],[],[]]
        Tg[2].append(temp[x]) #sets the goal state according the number of disks EG: [[],[],[4,3,2,1]]
    for x in xrange(1,pow(2,numberDisks)+1):
        for y in xrange(1,numberDisks+1):
            if pow(2,y)*(x)-pow(2,y-1)<pow(2,numberDisks):
                times[y-1].append(pow(2,y)*(x)-pow(2,y-1)) #sets the expected movements for each disk.

#Returns the last disk's value from 'tower' given
def lastDiskValue(tower):
    return Ti[tower][len(Ti[tower])-1]

#returns the expected direction movement of a given disk
def expectedDirection(disk):
    if disk%2==0 and numberDisks%2== 0:    #even number and even number of disk
        return -1                          #To LEFT
    elif disk%2!=0 and numberDisks%2== 0:  #Odd number and even number of disk
        return -2                          #To RIGHT
    elif disk%2==0 and numberDisks%2!= 0:  #even number and Odd number of disk
        return -2                          #To Right
    elif disk%2!=0 and numberDisks%2!= 0:  #Odd number and Odd number of disk
        return -1                          #To LEFT
    else:
        return -4                          #Failure case

#To check if a movement from "towerOrigin" to "towerDest" may be executed
def checkMovement(towerOrigin,towerDest):
    lastDiskOrigin=len(Ti[towerOrigin]) #Origin tower last disk's index
    lastDiskDest=len(Ti[towerDest])           #Destiny tower last disk's index

    if lastDiskOrigin ==0: #Checking if origin tower is empty
        return False
    else:
        if lastDiskDest==0 : #checking if destiny tower is empty
            return True
        elif lastDiskValue(towerOrigin)<lastDiskValue(towerDest):#checking if origin tower's last disk is less than
the destiny's
            return True
        else:
            return False

def printTi(fileToWrite):
    fileToWrite.write("\nTA: "+ str(Ti[0] ) + "\nTB: " + str(Ti[1] ) + "\nTC: " + str(Ti[2] )+ "\n")

#to show a possible disk's movement from 'originTower'  to 'destTower'. (it doesn't affect Ti)
def simulateMoveDish(originTower,destTower,heuris):
    Ti[originTower+destTower].append(Ti[originTower].pop()) #make movement
    generatedStates.write("\n\t\t\tHeuristics value: : "+str(heuris))
    generatedStates.write("\n\tTA: "+ str(Ti[0] ) + "\n\tTB: " + str(Ti[1] ) + "\n\tTC: " + str(Ti[2] )+ "\n")
    Ti[originTower].append(Ti[originTower+destTower].pop()) #returns movement

#to move a dish from 'originTower'  to 'destTower'
def moveDish(originTower,destTower):
    Ti[originTower+destTower].append(Ti[originTower].pop())
    printTi(solutionRoute)
    solutionRoute.write("_____________________\n")


# M A I N
numberDisks= int(input("Please, enter the number of disks:"))#ask for the number of disks to play
makeTimes()
print "Initial State: ",Ti

solutionRoute.write("\t\t\t Tower of Hanoi\n\n")
solutionRoute.write("\t\t Heuristics:  Best First\n")
solutionRoute.write("\t\t File: Solution Route\n\n")
solutionRoute.write("Initial State\n")
printTi(solutionRoute)
solutionRoute.write("_____________________\n")

generatedStates.write("\t\t\t Tower of Hanoi\n\n")
generatedStates.write("\t\t Heuristics:  Best First\n")
generatedStates.write("\t\t File: Generated States\n\n")

time_i = time()

while Ti!=Tg:

    for tower in range(0,len(Ti)):
        for displ in range(1,3):   #check if may displace -1 or -2 position in array, which means -1 ->1 to left    ; -2 -
> 1 to right
            if checkMovement(tower,tower-displ) == True:
                states[0].append(tower)    #add field 'towerOrigin'
                states[1].append(-displ)   #add field 'last disk's value' with the displacement
                states[2].append(0)        #add field 'heruistic' with initial value of 0



    for x in xrange(0,len(states[0])):
        if counterTime in times[lastDiskValue(states[0][x])-1]: #check if it's time to move disk x
            states[2][x]=states[2][x] + 1     #if yes, we add the minimum heuristic value possible .... +1
        else:
            states[2][x]=states[2][x] + 3     #if not, we add the maxium heuristic value possible .... +3

        if states[1][x] == expectedDirection(lastDiskValue(states[0][x])):
            states[2][x]=states[2][x] + 1
        else:
            states[2][x]=states[2][x] + 3

    generatedStates.write("\nMovement number: "+ str(counterTime))
    generatedStates.write("\nCurrent state of Towers:")
    printTi(generatedStates)
    generatedStates.write("\nPossible Movements:")

    for x in xrange(0,len(states[0])):
        simulateMoveDish(states[0][x],states[1][x],states[2][x])

    indexSelected=states[2].index(min(states[2]))

    moveDish(states[0][indexSelected],states[1][indexSelected])
    states=[[],[],[]]
    counterTime=counterTime+1
time_f = time()

time_ex = time_f - time_i

print "Final State",Ti

print 'Execution time:',time_ex,'seconds'
```

10 References

[1] Fuwan Ren, Qifan Yang, Jiexin Zheng, and Hui Yan. “Non-recursive Algorithm of Tower of Hanoi Problem”. 10th IEEE International Conference on Computer and Information Technology, pp. 2134 – 2137, 2010.

[2] Jun Wang1,2, Hong-fa Wang1, Guo-ying Yue1, and Nan Xie1. “Proving of the Non-recursive Algorithm for 4-Peg Hanoi Tower”. International Conference on Electronic Computer Technology, pp. 406-409, 2009.

[3] Giray Havur, Kadir Haspalamutgil, Can Palaz, Esra Erdem, and Volkan Patoglu. “A Case Study on the Tower of Hanoi Challenge: Representation, Reasoning and Execution”. IEEE International Conference on Robotics and Automation, pp. 4552- 4559, 2013.

[4] Steven L. Tanimoto. “Solving Problems by Drawing Solution Paths”. 2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VLlHCC), 2015.

[5] Huazhen Xu, Zhen You, and Jinyun Xue. “Automatic Verification of Non-recursive Algorithm of Hanoi Tower by Using Isabelle Theorem Prover*”. IEEE SNPD 2016, May 30-June 12016.

[6] Nadav Voloch, Elazar Birnbaum, and Amir Sapir. “Generating Error-Correcting Codes based on Tower of Hanoi Configuration Graphs”. IEEE 28-th Convention of Electrical and Electronics Engineers in Israel. 2014

[7] Rafiqul Islam, Shakib, Azizur Rahaman, Obaidur Rahman , and Al-Sakib Khan Pathan. “A Neighbour Discovery Approach for Cognitive Radio Network Using Tower of Hanoi (ToH) Sequence Based Channel Rendezvous”.

[8] https://gallery.technet.microsoft.com/scriptcenter/Towers-of-Hanoi-Recursive-8efcd412#content