CS5346 ADVANCED ARTIFICIAL INTELLIGENCE
PROJECT – 1 FALL 2023

SOLVING
TOWER OF HANOI GAME
USING A* SEARCH ALGORITHM AND RECURSIVE BEST FIRST ALGORITHM

Submitted by
Kavitha Lodagala (A05252367)

Team members:
Roopika Ganesh
Sakshi Tripathi

Under the guidance of
Dr. Moonis Ali

CONTENTS

1. PROBLEM DESCRIPTION

1.1 Description of the project

A classic puzzle or mathematics game is the Tower of Hanoi, also known as just Hanoi or the Tower of Hanoi. The French mathematician Edouard Lucas created it in 1883. Three rods and several disks of varying sizes that may be slid onto any rod make up the game. At the beginning of the puzzle, the disks are arranged neatly on a single rod in increasing size order, with the smallest disk at the top.

The Tower of Hanoi puzzle's goal is to transfer the whole disk stack from the beginning rod to a different rod while adhering to the following guidelines:

- You can only move one disk at a time.
- A disk can only be positioned on top of an empty rod or another larger disk.
- A smaller disk cannot be stacked on top of another disk.

The Tower of Hanoi puzzle can be solved with as little as 2n-1 movements, where "n" is the total number of disks. This issue has been researched in computer science and mathematics, with applications in recursion and algorithm design.

The Tower of Hanoi puzzle may be both enjoyable and difficult to solve, and it is frequently used to teach recursive algorithmic ideas. The Tower of Hanoi puzzle has several varieties with varied numbers of disks, making it less difficult to complete.

Recursive Best-First Search (RBFS) and the A* (A-star) algorithm are two effective and potent artificial intelligence systems. It estimates the cost of getting to the goal from a given stage using a heuristic function. The number of disks that are misaligned on the target peg is estimated by this heuristic. It is assumed that as the state gets closer to the goal state, fewer disks are misplaced. By decreasing the number of further searches and possibly improving the efficiency of achieving the objective state, the heuristic aids A* in prioritizing the most promising paths. A well-thought-out heuristic can more effectively direct the search in the Tower of Hanoi with additional disks by giving priority to paths that are most likely to result in the most accurate solutions.

1.2 Objective of Project

Our project goal is to use A* search and Recursive Best-First Search (RBFS) to solve the Tower of Hanoi puzzle. This probably entails putting these two search algorithms to use and evaluating how well they work to identify the optimal solutions. putting into practice the Tower of Hanoi problem representation, which includes the valid moves, goal state, initial state. Apply both the Recursive Best-First Search and the A* search algorithms.

Designing and implementing the heuristic function for RBFS and A* search. The cost of moving from the present state to the desired state should be estimated by these heuristic functions. Comparing the A* search and RBFS optimality results. Analyzing the both algorithms by comparing the time and space complexity. Additionally comparing the number

of nodes generated and expanded. Examine the relationship between these measurements and the Tower of Hanoi puzzle's disk count. It is anticipated that this research will improve knowledge of search algorithms, heuristic design, and how they are applied to solve practical issues. It also offers a chance to test and contrast various methods of problem-solving inside the framework of a conventional puzzle.

2. DOMAIN

2.1 Tower of Hanoi Game

The Tower of Hanoi is a classic mathematical puzzle that involves three pegs and a number of disks of different sizes. At the beginning of the puzzle, the disks are arranged neatly in size order on a single peg, with the smallest disk at the top. The goal is to transfer the complete stack to a different peg while adhering to these basic guidelines:

- You can only move one disk at a time.
- The upper disk from one stack is removed for each move, and it is then placed either on top of another stack or on an empty peg.
- A smaller disk cannot be stacked on top of another disk.

Usually, the pegs are assigned letter names that serve as descriptions of how the disks travel in relation to the pegs. For the three pegs, the typical protocol is to utilize the letters A, B, and C. For each peg name, the following is a brief explanation:

- A (Source Peg): The Tower of Hanoi puzzle begins at this peg. At first, every disk is arranged on this peg in increasing size order.
- B (Target Peg): The peg B, also known as the target peg, is where you wish to transfer the entire structure. Once every disk has been successfully shifted to this peg, the problem is deemed solved.
- C (Auxiliary Peg): During the solution procedure, this auxiliary or intermediate peg is utilized. As part of the approach for solving puzzles, disks may be temporarily relocated to this peg.

As part of the approach for solving puzzles, disks may be temporarily relocated to this peg. As an illustration, the tower of Hanoi's beginning state with n=3 is shown below. If it is necessary to relocate all disks from source peg (A) to target peg (B) and [3,2,1] are disks arranged in an ascending stack.

A -> [3,2,1] B -> [] C -> []

Auxiliary peg is used to shift all disks from A to B in order to achieve the desired state, as seen below.

A -> [] B -> [3,2,1] C -> []

One classic example that aids in the development of algorithmic thinking is the Tower of Hanoi issue. The problem must be divided into smaller subproblems, each of which must be solved systematically, in order to be solved. This method of approaching issue solving is essential in programming and computer science.

Understanding the limitations and structure of the problem is necessary to solve the Tower of Hanoi. It aids in the development of abilities connected to the mathematical or computational modeling of real-world issues, a crucial ability in computer science and engineering.

It calls for thorough preparation and analytical thought. It promotes strategic thinking about the steps required to reach the objective, which helps people develop their problem-solving abilities.

3. METHODOLOGIES

3.1 A* Search Algorithm

In computer science and artificial intelligence, the A* (pronounced "A-star") search method is a well-liked and frequently applied pathfinding technique. It is also called the best first search. In a graph or grid, it is used to discover the shortest path that accounts for the cost of traveling from one node to another in order to reach a goal. Since A* is an informed search algorithm, it bases its search on heuristic data. The reason A* is so effective and frequently used is that it combines the advantages of greedy search and uniform cost search.

The evaluation function f(n), which is the sum of g(n) and h(n), is used by the A* search to assess each node. where h(n) is the cost to travel from node n to the goal node and g(n) is the cost to go from root node to node n. When using an admissible heuristic function, A* is optimal since it never overestimates the cost of achieving the objective.

During the search, nodes are stored and retrieved using a priority queue. The f(n) values of the nodes determine their rank in the queue; lower values correspond to higher priority. Usage of a min-heap data structure is common. The "open set" of nodes that need to be evaluated and the "closed set" of nodes that have already been assessed are the two sets of nodes that A* keeps track of. Throughout the search procedure, nodes are transferred between these sets.

An overview of the A* algorithm is provided below:

- Use the starting node to initialize the open set.
- Even though the open set is not null:
- Take out of the open set the node with the lowest f(n) value.
- Rebuild and return the path if this node is the target.
- In any other case, add the neighbors of the node to the open set by calculating g(n), h(n), and f(n) for each neighbor.
- Transfer the node in use to the closed set.

The heuristic function's quality determines A*'s efficiency. An effective heuristic can drastically lower the total number of nodes investigated, which makes A* an effective pathfinding method for a variety of applications, including route planning, robotics, and video games.

3.2 Recursive Best First Search:

Recursive Best First Search (RBFS) is the best first search algorithm as well. It employs a recursive methodology and concentrates on examining the most promising paths based on the evaluation function.

The algorithm keeps track of a bound, or the lowest possible cost of the optimal alternative path from the current state. It examines nodes up to this cost limit. RBFS uses a recursive method to avoid keeping the complete search tree in memory. It monitors the top options while exploring the most promising paths.

RBFS is not always the best option, particularly when there is a requirement to dynamically modify the cost bound while searching. The particulars of the situation and the heuristic's quality determine how effective it is.

RBFS's main benefit is its capacity to carry out a best-first search with less memory use. Using a bound and a recursive method, RBFS prevents the search tree from being kept entirely in memory. Rather, it tracks the greatest options while investigating the most optimistic paths.

When memory is an issue, recursive best-first search comes in handy. However, it's crucial to strike a balance between pursuing potentially fruitful avenues and minimizing memory usage. In terms of temporal complexity, RBFS might not be effective compared to other algorithms, and its effectiveness is dependent on the specific characteristics of the problem and the quality of the heuristic employed in the evaluation function.

3.3 Heuristic Function

A heuristic function is a function in a search problem that calculates the value or cost of reaching a goal from a given state.

If a heuristic never overestimates the actual cost or value to achieve the goal state, it is considered admissible. Using an admissible heuristic, the search algorithm searches nodes in increasing order of estimated cost, ensuring that it will always locate the most effective solution.

In artificial intelligence, a heuristic function acts as a guide for search algorithms, offering approximations that assist in prioritizing state exploration in a manner that is most likely to result in an effective solution. It is common for the construction of efficient heuristics to require balancing computing efficiency and accuracy.

In this project for solving the Tower Hanoi game, Following are the three heuristic functions used in both A* and RBFS algorithms.

Heuristic_kavitha function 1:

This heuristic function considers the size and configuration of rings on the three rods (source, target, and auxiliary) to compute an estimate h(n) for the Tower of Hanoi problem. Each ring's

location and size on each rod are considered while determining cost. The size is considered because larger rings might be more important to be in the correct order than smaller rings, so penalize more for larger rings that are not in the correct order. The function sums up these values for all rods to provide an estimate of the cost to reach the goal state, guiding search algorithms like A*, Recursive best first search towards a solution.

```
def heuristic_kavitha(self):
        hn=0
        sn=len(self.source_rings)

        # iterates through each ring, adding values based on its position (distance from
the top) and its size
        for i in range(len(self.source_rings)):
            hn+=sn-i-1
            hn+=self.source_rings[i]

        tn=len(self.target_rings)

        #checks if each ring is not in the correct position, and if so, it adds values
based on both the position and size of the ring
        for i in range(len(self.target_rings)):
            if(self.n-i-self.target_rings[i]!=0):
                hn+=tn-i-1
                hn+=self.target_rings[i]

        an=len(self.auxiliary_rings)

        #iterates through each ring, adding values based on its position (distance from the
top) and its size
        for i in range(len(self.auxiliary_rings)):
            hn+=an-i-1
            hn+=self.auxiliary_rings[i]

        return hn
```

For the source rod and auxiliary rod, the function iterates through each ring, adding values based on its position (distance from the top) and its size. This considers both the order and size of the rings on the source and auxiliary rod.

For the target rod, the function checks if each ring is not in the correct position, and if so, it adds values based on both the position and size of the ring. This penalizes rings that are not in their correct order on the target rod.

The total of these values for each of the three rods is the final heuristic value (hn). The aim of this is to direct the search algorithm, considering the sizes and placements of the rings on each rod in the Tower of Hanoi problem, towards states that have a higher probability of leading to the destination state.

Heuristic_sakshi function 2:
This approach uses a heuristic function to compute the heuristic value, which is then supplied into the computation of the evaluation function. The weighted sum of the sizes—that is, the number on the disk—that are not on the destination rod is determined by this heuristic function. Since disks with higher numbers add more to the total, the function's goal is to move the disks to the target rod first while using the algorithm to solve the Tower of Hanoi issue.

```
    # this heuristic calculates the weighted sum of the number of discs not on the
destination rod
    def heuristic_sakshi(self):
        tn = len(self.target_rings)
        sum = 0
        for i in range(1, self.n + 1):
            if not (i in self.target_rings): sum += i
        return sum
```

Heuristic_roopika function 3:
This heuristic function computes the mismatched disk location and vacant slots in order to assess the degree of dissimilarity between the target peg and the aim. h(n) = mismatched components + empty spaces.

```
    def heuristic_roopika(self,reference_sequence):
        hn=0
        #reference_sequence =

        # Check if the array is empty
        if not self.target_rings:
            return 3  # Return 3 for an empty array

        # Check if the order of any one element in the input array matches the reference
array
        for i in range(len(self.target_rings)):
            if self.target_rings[i] == reference_sequence[i]:
                # Calculate the number of empty slots and mismatched elements in comparison
to the reference array
                empty_slots = len(reference_sequence) - len(self.target_rings)
                mismatched_elements = sum(1 for x, y in zip(self.target_rings,
reference_sequence) if x != y)
                hn = mismatched_elements + empty_slots
                return hn

        hn =3
        return hn
```

3.4 Evaluation Function

An essential part of the algorithm that aids in choosing which nodes to explore next is the evaluation function. The cost to reach a node from the beginning (g(n)) and the heuristic estimate of the cost to achieve the goal from that node (h(n)) are combined in a*. F(n)=g(n)+h(n) is the definition of the evaluation function f(n).

An overview of these elements is provided below:

- The g(n) is the true cost of transportation from root node to node n. It is the total cost incurred traveling to that node.
- h(n) this is the cost estimate using a heuristic method from node n to the goal. It offers an optimistic estimate of the remaining costs. Most significantly, h(n) should be admissible, which means that it never overestimates the actual cost to reach the goal.
- The function of evaluation, f(n). It combines the heuristic estimate (h(n)) with the actual cost (g(n)). For exploration, nodes with lower f(n) values are given priority.

3.5 Disadvantages of Existing System

A simple and effective solution to the Tower of Hanoi problem is to use a traditional recursive algorithm. In the Tower of Hanoi issue, a series of disks must be moved between rods while adhering to the following guidelines:

- You can only move one disk at a time.
- Only disks that are smaller than the disks currently on a rod can be transferred to the top of that rod.
- A smaller disk cannot be stacked on top of another disk.

Generally, the traditional recursive method works well with small to moderately sized disk counts. However, when the number of disks rises, its drawbacks become more noticeable.

The recursive method has an $O(2^n)$ time complexity, where n is the number of disks. This indicates that the time required to discover a solution increases quickly for increasing values of n. Practically speaking, this may result in longer execution times and not be appropriate for many disks. The recursive method depends on function calls, and the depth of recursion can get substantial for many disks. This could lead to a stack overflow, particularly if the maximum recursion depth or stack size in the programming environment.

Although the traditional recursive approach is conceptually straightforward and works best for the tower of Hanoi, it may not be as useful in other real-world situations due to its exponential time complexity and lack of precise path information, especially when handling bigger instances of the problem.

3.6 Proposed System

When solving the Tower of Hanoi using traditional recursive procedure is usually used to transfer the disks from one rod to another while adhering to the puzzle's principles. Without explicitly examining other options, the recursive approach guarantees the outcome by applying the Tower of Hanoi's mathematical laws directly.

Whereas, A* ensures completeness and optimality. It always discovers the best option at the lowest cost by methodically exploring the state space and considering many routes to the goal. The heuristic functions that direct the search and help in prioritizing pathways that are more likely to lead to the goal might be included to achieve the best possible solution.

4 SOURCE CODE IMPLEMENTATION

This project is implemented in python programming. A* Search Algorithm is implemented in Project2_A05252367_AStar.py file. Recursive Best first search algorithm is implemented in Project2_A05252367_RBFC.py file. In order to construct a tree we have used classes which is TowerOfHanoi and this class will have a constructor which creates each node.

4.1 A* Search code implementation:

The A* search algorithm is implemented in python by declaring a class TowerHanoi which contains the constructor (__init__()), build_hanoi_tree(), generate_child_nodes, check_in_visited(), print_tower_of_hanoi(), get_fn().

1. Constructor(init):It is used to create a node which contains the variables which need to be initialized.

```
    __init__(self,n,source_peg, source_rings, target_peg,
target_rings,auxiliary_peg,auxiliary_rings,level)
```

Here, n – number of disks
Source_peg – pole 'A'
Source_rings- rings present in source_peg
Targetpeg - pole 'B'
Target_rings- rings present in target_peg
Auxiliary_peg – pole 'C'
Auxiliary_rings - rings present in auxiliary_peg
Level – level each node. If a node(n) at level i then the children of node n will have level as i+1

2. Build_hanoi_tree(): This function contains the implementation of A* search algorithm. Firstly, closed, and open are lists where open is initialized with root node and closed is empty list initially. Every time the open is sorted and left most nodes are extracted which is called the best node. Then the algorithm checks whether the best node is the goal node. If the best node is not a goal node, then expand the best node and the child nodes are added to open. This process continues until a goal node is found or the open list becomes empty.
3. generate_child_nodes(): This function generates the child nodes based on the number of moves possible for a particular node.
4. Check_in_visited(): This function checks whether a node is already in visited nodes or not if a node already in visited then function returns true else returns false.
5. Print_tower_of_hanoi(): This function prints the rings present in source peg, target peg and auxiliary peg.
6. Get_fn(): This function calculates the evaluation function which is f(n), where f(n) = g(n) + h(n).

4.2 Recursive Best First Search Algorithm:

The Recursive Best First search algorithm is implemented in python by declaring a class TowerOfHanoi which contains the constructor (__init__()), goal_test(), Recursive_best_first_search(), RBFS_TowerOfHanoi(), generate_child_nodes, check_in_visited(), print_tower_of_hanoi(), get_fn().

1. Constructor(init):It is used to create a node which contains the variables which need to be initialized.

```
__init__(self,n,source_peg, source_rings, target_peg, target_rings, auxiliary_peg, auxiliary_rings, level)
```

Here, n – number of disks
Source_peg – pole 'A'
Source_rings- rings present in source_peg
Targetpeg - pole 'B'
Target_rings- rings present in target_peg
Auxiliary_peg – pole 'C'
Auxiliary_rings - rings present in auxiliary_peg
Level – level each node. If a node(n) at level i then the children of node n will have level as i+1

2. Goal_test: This function is used to test if a given node is a goal node or not. If the node is a goal node then returns true else returns false.
3. Recursive_best_first_search: This function takes the initial node as input parameter and passes it to RBFS_TowerOfHanoi. This function returns what RBFS_TowerOfHanoi returns.
4. RBFS_TowerOfHanoi: This function takes the initial node and f_limit as input. I have taken f_limit 10000 as the fn of any with disk count 3 to 11. At first the node will be tested against the goal node. If the node is goal then function returns success and fn value of the node. Otherwise, the node will be expanded and child nodes are added to the successors. Next, if the successor is empty then failure is returned, else the best node is found from the successor the node which has least fn and alternative node which is the next least fn value. Now the function is recursively called and stops once a goal node is achieved.
5. generate_child_nodes(): This function generates the child nodes based on the number of moves possible for a particular node.
6. Check_in_visited(): This function checks whether a node is already in visited nodes or not if a node already in visited then function returns true else returns false.
7. Print_tower_of_hanoi(): This function prints the rings present in source peg, target peg and auxiliary peg.

The main important python packages used are as follows-

- time package which is used to calculate the time for executing the program.
- Matplotlib package is used to plot the nodes generated vs number of disks. This helps to analyze the program output. And also we can analyze the heuristic function efficiency by comparing all 3 heuristics results like execution time, number of nodes generated and number of nodes extended.
- Psutil package to calculate the memory allocated for both the algorithms.
  memory = psutil.Process().memory_info().rss / (1024 * 1024)
  memory is calculated by using Process class and it contains a member function memory_info. In order get the memory in Megabytes the result is divided by 1024*1024
- Tabulate package is used to display the results in tabular format.

- Stack is used to pop the least f(n) value and sort function is used to sort the nodes based on f(n) value.
- Sort function is used to sort the open in A* search algorithm.

5 SOURCE CODE

Project is implemented in python programming

5.1 A* Search Algorithm:

Project2_A05252367_AStar.py

```
"""
****************** File Name: Project1_A05252367_Astar.py *********************
# This file contains the implementation of A* search algorithm
# TowerOfHanoi is a class which can be used to build the tree
# Possible moves of particular node will be generated as child node
# Once a node matches the goal node then loop breaks
# This process repeats for all the disks starting from
# n=3 to 11( which complete in 30 Mins)
*******************************************************************************
"""

from itertools import product
import time
import matplotlib.pyplot as plt
import psutil
from tabulate import tabulate

class TowerOfHanoi:

    '''
    ************************Constructor*********************************
    # In this we initialize all the variables required for each node.
    # This function is called whenever a node is created
    # Here the source peg is 'A', Target peg is 'B', Auxiliary peg id 'C'
    # source rings contains rings containing in source peg
    # target rings contains rings containing in target peg
    # Auxiliary rings contains rings containing in Auxiliary peg
    # gn is the level of node in the tree, which is level of parent node + 1
    # hn is the heuristic value of a node
    ********************************************************************
    '''
    def __init__(self,n,source_peg, source_rings, target_peg,
target_rings,auxiliary_peg,auxiliary_rings,level):
        self.n = n  # Number of disks
        self.source_peg = source_peg  # Source peg
        self.source_rings=source_rings # Rings present in source peg
        self.target_peg = target_peg  # Target peg
        self.target_rings = target_rings # Rings present in target peg
        self.auxiliary_peg = auxiliary_peg  # Auxiliary peg
        self.auxiliary_rings = auxiliary_rings  # Rings present in auxiliary peg
        self.gn=level # level of the tree which g(n)
        #self.hn=self.heuristic_roopika(expected_rings_order) # heuristic function1
        #self.hn=self.heuristic_sakshi() # heuristic function2
        self.hn=self.heuristic_kavitha() # heuristic function3
        self.child = []

    '''
```

```
    ******************Function Name: heuristic_roopika**************
    # This function takes number of disks as input parameter
    #
    # Returns the heuristic value
    ****************************************************************
    '''
    def heuristic_roopika(self,reference_sequence):
        hn=0
        #reference_sequence =

        # Check if the array is empty
        if not self.target_rings:
            return 3  # Return 3 for an empty array

        # Check if the order of any one element in the input array matches the reference
array
        for i in range(len(self.target_rings)):
            if self.target_rings[i] == reference_sequence[i]:
                # Calculate the number of empty slots and mismatched elements in comparison
to the reference array
                empty_slots = len(reference_sequence) - len(self.target_rings)
                mismatched_elements = sum(1 for x, y in zip(self.target_rings,
reference_sequence) if x != y)
                hn = mismatched_elements + empty_slots
                return hn

        hn =3
        return hn


    '''
    ****************Function Name: heuristic_kavitha *****************
    # This heuristic function calculates an estimate (hn) for the Tower of Hanoi problem by
considering the number
    # and arrangement of rings on the three rods (source, target, and auxiliary).
    ****************************************************************
    '''
    def heuristic_kavitha(self):
        hn=0

        sn=len(self.source_rings)# number of rings in source peg

        # iterates through each ring, adding values based on its position (distance from
the top) and its size
        for i in range(len(self.source_rings)):
            hn+=sn-i-1
            hn+=self.source_rings[i]

        tn=len(self.target_rings)# number of rings in target peg

        #checks if each ring is not in the correct position, and if so, it adds values
based on both the position and size of the ring
        for i in range(len(self.target_rings)):
            if(self.n-i-self.target_rings[i]!=0):
                hn+=tn-i-1
                hn+=self.target_rings[i]

        an=len(self.auxiliary_rings)# number of rings in auxiliary peg
```

```
        #iterates through each ring, adding values based on its position (distance from the
top) and its size
        for i in range(len(self.auxiliary_rings)):
            hn+=an-i-1
            hn+=self.auxiliary_rings[i]

        return hn


    '''
    ****************** Function Name: heuristic_sakshi ******************
    # this heuristic calculates the weighted sum of the number of discs not on the
destination rod
    ***********************************************************************
    '''
    def heuristic_sakshi(self):
        tn = len(self.target_rings)
        sum = 0
        for i in range(1, self.n + 1):
            if not (i in self.target_rings): sum += i
        return sum


    '''
    ******************Function Name: get_fn *********************
    # returns evaluation function values that sum of gn and hn
    # f(n) = g(n) + h(n)
    # returns the fn value
    ************************************************************
    '''
    def get_fn(self):
        return self.gn+self.hn

    '''
    ***************** Function Name: print_tower_of_hanoi *****************
    # This function prints the present state of the node which means rings
    present in source peg, target peg and auxiliary peg
    ***********************************************************************
    '''
    def print_tower_of_hanoi(self):
        print("g(n):",self.gn," h(n):",self.hn," f(n)=",self.gn+self.hn)
        print(self.source_peg,"-->",self.source_rings)
        print(self.target_peg,"-->",self.target_rings)
        print(self.auxiliary_peg,"-->",self.auxiliary_rings)


    '''
    ***************** Function Name: check_in_visited **********************
    # This function checks whether a node is visited or not
    # If a node is visited then return true
    # Otherwise it adds the node to visited_nodes list and returns flase
    ************************************************************************
    '''
    def check_in_visited(self):
        global visited_nodes
```

```
        s=self.source_peg+''.join(list(map(str,self.source_rings)))+self.target_peg+''.join
(list(map(str,self.target_rings)))+self.auxiliary_peg+''.join(list(map(str,self.auxiliary_r
ings)))

        # check whether the node is presen in visited nodes or not
        if(s not in visited_nodes):
            # If node not in visited then add nodes to the visited
            visited_nodes.append(s)
            return False
        else:
            # Node already visited
            return True

    '''
    ****************** Function Name: generate_child_nodes *****************
    # This function generate the childrens of a given node
    ************************************************************************
    '''
    def generate_child_nodes(self,best_node):
        global visited_nodes
        global count

        sn=len(best_node.source_rings)# number of rings in source peg
        tn=len(best_node.target_rings)# number of rings in target peg
        an=len(best_node.auxiliary_rings)# number of rings in auxiliary peg

        # Checking whether source peg has any rings
        if(sn>0):
            # creating a new node by moving top most ring from source peg to target peg
only if the source peg top ring
            # is smaller than the target top most ring or if target peg has no rings then
move ring from source directly
            if(tn<n and ((tn>0 and sn>0 and best_node.source_rings[-
1]<best_node.target_rings[-1]) or tn==0)):
                ring=best_node.source_rings[-1]

                #creating a child node with new move
                child_node = TowerOfHanoi(n,best_node.source_peg,
best_node.source_rings[:sn-1], best_node.target_peg,
best_node.target_rings+[ring],best_node.auxiliary_peg,best_node.auxiliary_rings,best_node.g
n+1)
                best_node.child.append(child_node)

                # checking if the child node visited or not if  visited then ignored
otherwise child node is added to open
                if(not child_node.check_in_visited()):
                    open.append([child_node,states[count],child_node.get_fn()])
                    count+=1

            # creating a new node by moving top most ring from source peg to auxiliary peg
only if the source peg top ring
            # is smaller than the auxiliary top most ring or if auxiliary peg has no rings
then move ring from source directly
            if(an<n and ((an>0 and sn>0 and best_node.source_rings[-
1]<best_node.auxiliary_rings[-1]) or an==0)):
                ring=best_node.source_rings[-1]

                #creating a child node with new move
```

```
                child_node = TowerOfHanoi(n,best_node.source_peg,
best_node.source_rings[:sn-1], best_node.target_peg,
best_node.target_rings,best_node.auxiliary_peg,best_node.auxiliary_rings+[ring],best_node.g
n+1)
                best_node.child.append(child_node)

                # checking if the child node visited or not if  visited then ignored
otherwise child node is added to open
                if(not child_node.check_in_visited()):
                    open.append([child_node,states[count],child_node.get_fn()])
                    count+=1

        # Checking whether target peg has any rings
        if(tn>0):
            # creating a new node by moving top most ring from target peg to source peg
only if the target peg top ring
            # is smaller than the source top most ring or if source peg has no rings then
move ring from target directly
            if(sn<n and ((sn>0 and tn>0 and best_node.target_rings[-
1]<best_node.source_rings[-1]) or sn==0)):
                ring=best_node.target_rings[-1]

                #creating a child node with new move
                child_node= TowerOfHanoi(n,best_node.source_peg,
best_node.source_rings+[ring], best_node.target_peg, best_node.target_rings[:tn-
1],best_node.auxiliary_peg,best_node.auxiliary_rings,best_node.gn+1)
                best_node.child.append(child_node)

                # checking if the child node visited or not if  visited then ignored
otherwise child node is added to open
                if(not child_node.check_in_visited()):
                    open.append([child_node,states[count],child_node.get_fn()])
                    count+=1

            # creating a new node by moving top most ring from target peg to auxiliary peg
only if the target peg top ring
            # is smaller than the auxiliary top most ring or if auxiliary peg has no rings
then move ring from target directly
            if(an<n and ((an>0 and tn>0 and best_node.target_rings[-
1]<best_node.auxiliary_rings[-1]) or an==0)):
                ring=best_node.target_rings[-1]

                #creating a child node with new move
                child_node= TowerOfHanoi(n,best_node.source_peg, best_node.source_rings,
best_node.target_peg, best_node.target_rings[:tn-
1],best_node.auxiliary_peg,best_node.auxiliary_rings+[ring],best_node.gn+1)
                best_node.child.append(child_node)

                # checking if the child node visited or not if  visited then ignored
otherwise child node is added to open
                if(not child_node.check_in_visited()):
                    open.append([child_node,states[count],child_node.get_fn()])
                    count+=1

        # Checking whether auxiliary peg has any rings
        if(an>0):
```

```
            # creating a new node by moving top most ring from auxiliary peg to source peg
only if the auxiliary peg top ring
            # is smaller than the source top most ring or if source peg has no rings then
move ring from auxiliary directly
            if(sn<n and ((sn>0 and an>0 and best_node.auxiliary_rings[-
1]<best_node.source_rings[-1]) or sn==0)):
                ring=best_node.auxiliary_rings[-1]

                #creating a child node with new move
                child_node= TowerOfHanoi(n,best_node.source_peg,
best_node.source_rings+[ring], best_node.target_peg,
best_node.target_rings,best_node.auxiliary_peg,best_node.auxiliary_rings[:an-
1],best_node.gn+1)
                best_node.child.append(child_node)

                # checking if the child node visited or not if  visited then ignored
otherwise child node is added to open
                if(not child_node.check_in_visited()):
                    open.append([child_node,states[count],child_node.get_fn()])
                    count+=1

            # creating a new node by moving top most ring from auxiliary peg to target peg
only if the auxiliary peg top ring
            # is smaller than the target top most ring or if target peg has no rings then
move ring from auxiliary directly
            if(tn<n and ((tn>0 and an>0 and best_node.auxiliary_rings[-
1]<best_node.target_rings[-1]) or tn==0)):
                ring=best_node.auxiliary_rings[-1]

                #creating a child node with new move
                child_node= TowerOfHanoi(n,best_node.source_peg, best_node.source_rings,
best_node.target_peg,
best_node.target_rings+[ring],best_node.auxiliary_peg,best_node.auxiliary_rings[:an-
1],best_node.gn+1)
                best_node.child.append(child_node)

                # checking if the child node visited or not if  visited then ignored
otherwise child node is added to open
                if(not child_node.check_in_visited()):
                    open.append([child_node,states[count],child_node.get_fn()])
                    count+=1

    '''
    ***************** Function Name: print_open_closed ******************
    # This function prints nodes present in open and closed
    *********************************************************************
    '''

    def print_open_closed(self,open,closed):
            print("**********Printing Open List and Closed List*********")
            print("\nOpen-->[",end="")
            for i in range(len(open)):
                print(open[i][1]+str(open[i][2]),end=" ")
            print("]")

            print("Closed-->[",end="")
            for i in range(len(closed)):
                print(closed[i][1]+str(closed[i][2]),end=" ")
```

```
            print("]")


    '''
    ****************** Function Name: build_hanoi_tree ********************
    # This function does the actual A* search
    # Initially root node added to open and loop continue to expand the node with less fn
value
    # The node with least fn value is the best node
    # If the best node is goal node then loop breaks
    # Otherwise best node is expanded by calling the function generate child nodes
    # Returns number of nodes generated and number of nodes expanded.
    ***********************************************************************
    '''
    def build_hanoi_tree(self):

        # using global count to store the number of nodes generated
        global count

        # Closed list stores the nodes that are expanded
        closed=[]

        # best_node stores the node that is poped from open that is the no  de with less fn
value
        best_node=None

        #adding the current node to the visited by calling check_in_visited() function name
        s=self.check_in_visited()

        # loop runs until open becomes empty or a goal node is found
        while(open):

            # sorting the open based on fn value
            open.sort(reverse=True,key=lambda x:x[2])

            # self.print_open_closed(open,closed)  #this prints the tracing of the open and
closed


            # best node will be stored in best_node
            best_node=open.pop()

            # best node will be added to closed
            closed.append(best_node)

            best_node=best_node[0]

            #print("****** Selected Best node ****** " )
            #best_node.print_tower_of_hanoi()

            #checking whether the best node is goal node
            if((best_node!=None and best_node.target_rings==expected_rings_order)):
                print("Disks are placed in target pole B")
                best_node.print_tower_of_hanoi()
                break

            # expanding the best node
            self.generate_child_nodes(best_node)
```

```
        print("total number of nodes generated : ", len(open)+len(closed), "Total number of
Best nodes expanded : ", len(closed))


        return (len(open)+len(closed),len(closed))



n = 3  # Number of rings  and n starts with 3
No_of_disks=[] # stores number of disk of tower hanoi that are solved
Node_Generated=[] # stores number of nodes generated for each disk count
Node_Expanded=[] # stores number of nodes expanded for each disk count

# In order to give each state a name like ('A','B','C'...) below loop is run to get
distinct state names
alp=list(map(chr, range(65, 91)))

# To get a combination of alphabets from a single letter to 6 letters
states=[]
for i in range(1,6):
    for comb in product(alp, repeat=i):
        states.append(''.join(comb))

# table_data list is to store the number of disks, elapsed time, memory used,
# nodes generated and nodes expanded in a tabular formate using tabulate package
table_data=[]
table_data.append(["Number of Disks","Elapsed Time","Memory Used","Nodes Generated","Nodes
Expanded"])

# A* search algorithm solves tower hanoi for 3 to 11 disks in 30 mins
# so running loop for n starting from n=3 to 11
while(n<12):
    print("\nstarting A* algorithm for",n, "disks")
    count = 0 # count stores number of nodes generated

    #To set the expected order of rings on the target peg and source peg
    expected_rings_order=list(range(n,0,-1))

    #Set the root node by calling the class and initializing them
    root = TowerOfHanoi(n,'A', expected_rings_order, 'B', [],'C',[],0)

    #add the root to the open list along with state and f(n) value of the root node
    open=[[root,states[count],root.get_fn()]]

    count+=1

    # visited nodes contains the nodes that are generated
    visited_nodes=[]

    # Record the start time
    start_time = time.time()

    #build the tower of hanoi tree using this root node.
    generated,expanded=root.build_hanoi_tree()

    # Record the end time
    end_time = time.time()
```

```
    # Calculate the elapsed time
    elapsed_time = end_time - start_time

    # Memory allocated for each disk count will be calculated and added to table_data
    memory = psutil.Process().memory_info().rss / (1024 * 1024)

    # storing the number of nodes generated and number of nodes expanded for each disk
count
    Node_Generated.append(generated)
    Node_Expanded.append(expanded)

    # table_data is used to tabulate all the details for each disk
    table_data.append([n,elapsed_time,memory,generated,expanded])

    # Print the elapsed time
    print(f"Elapsed time: {elapsed_time} seconds")

    # Print the memory consumed
    print("Memory consumed : ", memory, "MB")

    # adding the disk count for plotting
    No_of_disks.append(n)

    # increamenting the disk count to continue solving of Tower Hanoi using A* search
    n+=1

# Prints the all disk count and corresponding time, memory , nodes generated and expanded
print(tabulate(table_data))

# plotting graph to compare number of nodes generated and number of nodes expanded for each
disk count
x= No_of_disks
y1=Node_Generated
y2=Node_Expanded
plt.plot(x,y1, label = "no of disks VS no of nodes generated")
plt.plot(x,y2, label = "no of disks VS no of nodes expanded")

# Adding labels and title
plt.xlabel('Number of disks')
plt.ylabel('Number of nodes')
plt.title('A* Algorithm for Tower of Hanoi')
plt.legend()

# Display the plot
plt.show()
```

5.2 Recursive Best First Search Algorithm:

Project2_A05252367_RBFS.py

```
"""
******** File Name: Project1_A05252367_RBFS.py **************
```

```
# This file contains the implementation of Recurssive Best Firs Search algorithm
# TowerOfHanoi is a class which can be used to build the tree
# Possible moves of particular node will be generated as child node
# Once the child node is generated best successor and alternative best successors are
stored
# Now the best node child nodes are generated and if the best child node fn value is
greater than alternative then
# alternative node is expanded this process continues recursively until a goal node is
found
# This process repeats for all the disks starting from n=3 to 11( which complete in 30
Mins)
**************************************************************************
"""

from itertools import product
import time
import matplotlib.pyplot as plt
import psutil
from tabulate import tabulate


class TowerOfHanoi:

    '''
    ******************Constructor*****************************************
    # In this we initialize all the variables required for each node.
    # This function is called whenever a node is created
    # Here the source peg is 'A', Target peg is 'B', Auxiliary peg id 'C'
    # source rings contains rings containing in source peg
    # target rings contains rings containing in target peg
    # Auxiliary rings contains rings containing in Auxiliary peg
    # gn is the level of node in the tree, which is level of parent node +1
    # hn is the heuristic value of a node
    **************************************************************************
    '''
    def __init__(self,n,source_peg, source_rings, target_peg,
target_rings,auxiliary_peg,auxiliary_rings,level):
        self.n = n  # Number of disks
        self.source_peg = source_peg  # Source peg
        self.source_rings=source_rings # Rings present in source peg
        self.target_peg = target_peg  # Target peg
        self.target_rings = target_rings # Rings present in target peg
        self.auxiliary_peg = auxiliary_peg  # Auxiliary peg
        self.auxiliary_rings = auxiliary_rings # Rings present in auxiliary peg
        self.gn=level #level of the tree which g(n)
        #self.hn=self.heuristic_roopika(expected_rings_order)  # heuristic function1
        #self.hn=self.heuristic_sakshi()  # heuristic function2
        self.hn=self.heuristic_kavitha()  # heuristic function3
        self.fn=self.gn + self.hn # evaluation function values that sum of gn and hn f(n) =
g(n) + h(n)
        self.child=[]


    '''
    ******************Function Name: heuristic_roopika***************
    # This function takes number of disks as input parameter
    #
    # Returns the heuristic value
```

```
    ****************************************************************
    '''
    def heuristic_roopika(self,reference_sequence):
        hn=0
        #reference_sequence =

        # Check if the array is empty
        if not self.target_rings:
            return 3  # Return 3 for an empty array

        # Check if the order of any one element in the input array matches the reference
array
        for i in range(len(self.target_rings)):
            if self.target_rings[i] == reference_sequence[i]:
                # Calculate the number of empty slots and mismatched elements in comparison
to the reference array
                empty_slots = len(reference_sequence) - len(self.target_rings)
                mismatched_elements = sum(1 for x, y in zip(self.target_rings,
reference_sequence) if x != y)
                hn = mismatched_elements + empty_slots
                return hn

        hn =3
        return hn


    '''
    ****************Function Name: heuristic_kavitha *****************
    # This heuristic function calculates an estimate (hn) for the Tower of Hanoi problem by
considering the number
    # and arrangement of rings on the three rods (source, target, and auxiliary).
    ****************************************************************
    '''
    def heuristic_kavitha(self):
        hn=0

        sn=len(self.source_rings)# number of rings in source peg

        # iterates through each ring, adding values based on its position (distance from
the top) and its size
        for i in range(len(self.source_rings)):
            hn+=sn-i-1
            hn+=self.source_rings[i]

        tn=len(self.target_rings)# number of rings in target peg

        #checks if each ring is not in the correct position, and if so, it adds values
based on both the position and size of the ring
        for i in range(len(self.target_rings)):
            if(self.n-i-self.target_rings[i]!=0):
                hn+=tn-i-1
                hn+=self.target_rings[i]

        an=len(self.auxiliary_rings)# number of rings in auxiliary peg

        #iterates through each ring, adding values based on its position (distance from the
top) and its size
        for i in range(len(self.auxiliary_rings)):
```

```
            hn+=an-i-1
            hn+=self.auxiliary_rings[i]

        return hn

    '''
    ****************** Function Name: heuristic_sakshi ******************
    # this heuristic calculates the weighted sum of the number of discs not on the
destination rod
    **********************************************************************
    '''
    def heuristic_sakshi(self):
        tn = len(self.target_rings)
        sum = 0
        for i in range(1, self.n + 1):
            if not (i in self.target_rings): sum += i
        return sum

    '''
    ***************** Function Name: print_tower_of_hanoi *****************
    # This function prints the present state of the node which means rings
    present in source peg, target peg and auxiliary peg
    **********************************************************************
    '''
    def print_tower_of_hanoi(self):
        print("g(n):",self.gn," h(n):",self.hn," f(n)=",self.fn)
        print(self.source_peg,"-->",self.source_rings)
        print(self.target_peg,"-->",self.target_rings)
        print(self.auxiliary_peg,"-->",self.auxiliary_rings)


    '''
    ******************* Function Name: goal_test **********************
    # This function takes node as input and checks whether the current
    node is the goal node or not
    **********************************************************************
    '''
    def goal_test(self,node):
        #checking current node is the goal or not
        if(node.target_rings==list(range(self.n,0,-1))):
            return True
        else:
            return False

    '''
    ****************** Function Name: Recursive_best_first_search **********************
    # This function takes initial node as input parameter and calls RBFS_TowerOfHanoi
function
    # and returns the output
    ************************************************************************************
    '''
    def Recursive_best_first_search(self,initial_node):
        return self.RBFS_TowerOfHanoi(initial_node,10000)

    '''
    *************** Function Name: check_in_visited ***********************
    # This function checks whether a node is visited or not
    # If a node is visited then return true
```

```
    # Otherwise it adds the node to visited_nodes list and returns flase
    ************************************************************************
    '''
    def check_in_visited(self):
        s=self.source_peg+''.join(list(map(str,self.source_rings)))+self.target_peg+''.join
(list(map(str,self.target_rings)))+self.auxiliary_peg+''.join(list(map(str,self.auxiliary_r
ings)))

        # check whether the node is presen in visited nodes or not
        if(s not in visited_nodes):
            # If node not in visited then add nodes to the visited
            visited_nodes.append(s)
            return False
        else:
            # Node already visited
            return True

    def RBFS_TowerOfHanoi(self,node,f_limit):
        global expanded_nodes
        if(self.goal_test(node)):
            node.print_tower_of_hanoi()
            return "Success",node.fn
        successors=[]

        sn=len(node.source_rings)
        tn=len(node.target_rings)
        an=len(node.auxiliary_rings)
        #node.print_tower_of_hanoi()
        if(node not in expanded_nodes):
            expanded_nodes.append(node)
            if(sn>0):
                if(tn<n and ((tn>0 and sn>0 and node.source_rings[-1]<node.target_rings[-
1]) or tn==0)):
                    ring=node.source_rings[-1]
                    child_node = TowerOfHanoi(n,node.source_peg, node.source_rings[:sn-1],
node.target_peg,
node.target_rings+[ring],node.auxiliary_peg,node.auxiliary_rings,node.gn+1)
                    if(not child_node.check_in_visited()):
                        node.child.append([child_node,child_node.fn])
                        successors.append([child_node,child_node.fn])
                    #child_node.print_tower_of_hanoi()
                if(an<n and ((an>0 and sn>0 and node.source_rings[-
1]<node.auxiliary_rings[-1]) or an==0)):
                    ring=node.source_rings[-1]
                    child_node = TowerOfHanoi(n,node.source_peg, node.source_rings[:sn-1],
node.target_peg,
node.target_rings,node.auxiliary_peg,node.auxiliary_rings+[ring],node.gn+1)
                    if(not child_node.check_in_visited()):
                        node.child.append([child_node,child_node.fn])
                        successors.append([child_node,child_node.fn])
                    #child_node.print_tower_of_hanoi()
            if(tn>0):
                if(sn<n and ((sn>0 and tn>0 and node.target_rings[-1]<node.source_rings[-
1]) or sn==0)):
                    ring=node.target_rings[-1]
                    child_node= TowerOfHanoi(n,node.source_peg, node.source_rings+[ring],
node.target_peg, node.target_rings[:tn-1],node.auxiliary_peg,
node.auxiliary_rings,node.gn+1)
```

```
                    if(not child_node.check_in_visited()):
                        node.child.append([child_node,child_node.fn])
                        successors.append([child_node,child_node.fn])
                    #child_node.print_tower_of_hanoi()
                if(an<n and ((an>0 and tn>0 and node.target_rings[-
1]<node.auxiliary_rings[-1]) or an==0)):
                    ring=node.target_rings[-1]
                    child_node= TowerOfHanoi(n,node.source_peg, node.source_rings,
node.target_peg, node.target_rings[:tn-
1],node.auxiliary_peg,node.auxiliary_rings+[ring],node.gn+1)
                    if(not child_node.check_in_visited()):
                        node.child.append([child_node,child_node.fn])
                        successors.append([child_node,child_node.fn])
                    #child_node.print_tower_of_hanoi()
            if(an>0):
                if(sn<n and ((sn>0 and an>0 and node.auxiliary_rings[-
1]<node.source_rings[-1]) or sn==0)):
                    ring=node.auxiliary_rings[-1]
                    child_node= TowerOfHanoi(n,node.source_peg, node.source_rings+[ring],
node.target_peg, node.target_rings,node.auxiliary_peg,node.auxiliary_rings[:an-
1],node.gn+1)
                    if(not child_node.check_in_visited()):
                        node.child.append([child_node,child_node.fn])
                        successors.append([child_node,child_node.fn])
                    #child_node.print_tower_of_hanoi()
                if(tn<n and ((tn>0 and an>0 and node.auxiliary_rings[-
1]<node.target_rings[-1]) or tn==0)):
                    ring=node.auxiliary_rings[-1]
                    child_node= TowerOfHanoi(n,node.source_peg, node.source_rings,
node.target_peg, node.target_rings+[ring],node.auxiliary_peg,node.auxiliary_rings[:an-
1],node.gn+1)
                    if(not child_node.check_in_visited()):
                        node.child.append([child_node,child_node.fn])
                        successors.append([child_node,child_node.fn])
                    #child_node.print_tower_of_hanoi()
        else:
            expanded_nodes.append(node)
            successors=node.child
            for suc in successors:
                s=suc[0].source_peg+''.join(list(map(str,suc[0].source_rings)))+suc[0].targ
et_peg+''.join(list(map(str,suc[0].target_rings)))+suc[0].auxiliary_peg+''.join(list(map(st
r,suc[0].auxiliary_rings)))
                visited_nodes.append(s)
        if(successors==[]):
            #print("Failure",10000,f_limit)
            return "Failure", 10000

        for s in successors:
            s[0].fn=max(s[0].fn , node.fn)
            s[1]=s[0].fn

        while(1):
            successors.sort(reverse=True,key=lambda x:x[1])
            best_node=successors[-1]

            if(best_node[0].fn>f_limit):
                #print("Failure",best_node[0].fn,f_limit)
                return "Failure",best_node[0].fn
```

```
            if(len(successors)>=2):
                alternative_node=successors[-2]
                #print("best node")
                #best_node[0].print_tower_of_hanoi()
                #print("alternative node")
                #alternative_node[0].print_tower_of_hanoi()
                result,best_node[0].fn=self.RBFS_TowerOfHanoi(best_node[0],min(f_limit,alte
rnative_node[0].fn))
                best_node[1]=best_node[0].fn
            else:
                result,best_node[0].fn=self.RBFS_TowerOfHanoi(best_node[0],f_limit)
                best_node[1]=best_node[0].fn
            if(result!="Failure"):
                return result,best_node[0].fn
start_time1=time.time()


n = 3 # Number of rings  and n starts with 3
No_of_disks=[] # stores number of disk of tower hanoi that are solved
Node_Generated=[] # stores number of nodes generated for each disk count
Node_Expanded=[] # stores number of nodes expanded for each disk count

# table_data list is to store the number of disks, elapsed time, memory used,
# nodes generated and nodes expanded in a tabular formate using tabulate package
table_data=[]
table_data.append(["Number of Disks","Elapsed Time","Memory Used","Nodes Generated","Nodes
Expanded"])

while(n<10):
    #print(time.time()-start_time1,time.time()-start_time1>10)
    print("\nstarting Recursive best first search algorithm for",n, "disks")


    expanded_nodes=[]

    #To set the expected order of rings on the target peg and source peg
    expected_rings_order=list(range(n,0,-1))

    # visited nodes contains the nodes that are generated
    visited_nodes=[]

    #Set the root node by calling the class and initializing them
    root = TowerOfHanoi(n,'A', expected_rings_order, 'B', [],'C',[],0)

    start_time = time.time()

    # Set the root node by calling the class and initializing them
    result,f_value=root.Recursive_best_first_search(root)

    end_time = time.time()

    # Calculate the elapsed time
    execution_time = end_time - start_time

    # Memory allocated for each disk count will be calculated and added to table_data
    memory = psutil.Process().memory_info().rss / (1024 * 1024)

    # Print the memory consumed
```

```
    table_data.append([n,execution_time,memory,len(visited_nodes), len(expanded_nodes)])

    Node_Generated.append(len(visited_nodes))
    Node_Expanded.append(len(expanded_nodes))

    print("total number of nodes generated : ", len(visited_nodes), "Total number of Best
nodes expanded : ", len(expanded_nodes))

    # Print the elapsed time
    print(f"Elapsed time: {execution_time} seconds")

    # Print the memory consumed
    print("Memory consumed : ", memory, "MB")


    # adding the disk count for plotting
    No_of_disks.append(n)

    # increamenting the disk count to continue solving of Tower Hanoi using A* search
    n+=1

# Prints the all disk count and corresponding time, memory , nodes generated and expanded
print(tabulate(table_data))

# plotting graph to compare number of nodes generated and number of nodes expanded for each
disk count
x= No_of_disks
y1=Node_Generated
y2=Node_Expanded
plt.plot(x,y1, label = "no of disks VS no of nodes generated")
plt.plot(x,y2, label = "no of disks VS no of nodes expanded")

# Adding labels and title
plt.xlabel('Number of disks')
plt.ylabel('Number of nodes')
plt.title('RBFS Algorithm for Tower of Hanoi')
plt.legend()

# Display the plot
plt.show()
```

6 ANALYSIS OF THE PROGRAM

The main significance of this project is to develop an intelligent system which is used to solve the tower of Hanoi game of n number of disks. This is done by using two algorithms A* search and Recursive best first search. A* search is optimal but requires more memory than RBFS. RBFS is more memory efficient than A* search. This is because instead of storing the nodes in open and closed like in A*, RBFS stores only the best node and alternative node to improve memory usage. But this approach leads to node regeneration which consumes more time.

The following features are included to improve the program execution:

1. In A* search and RBFS, possible moves of rings among the 3 pegs are considered as child nodes. If a move is already generated then the same move in future nodes should

not be made; this improves to avoid unnecessary node expansion. This process is done by the check_in_visited function which returns true or false. If a node is already visited then the function returns the true otherwise it returns false.

2. In A* search algorithm, open is used as a stack as we can pop the elements from the stack more efficiently than using a simple list. So this has helped to access and play around the open.
3. In the heuristic_kavitha function, Instead of considering only the misplaced rings it also takes position and size of the rings into account to reach the goal state. This approach improved the efficiency of program execution and number of nodes generated.
4. In RBFS Algorithm, to avoid the regeneration of nodes. The algorithm is slightly modified by including an extra list (expanded_nodes) which contains the nodes already expanded and corresponding the successors are returned. This improves efficiency of RBFS by reducing the regeneration of child nodes and time consumption.
5. The sort function used in both the algorithms helped to sort the nodes based on the fn value of each node.
6. Whenever a constructor is called a new node is generated will the attributes all three pegs (A,B,C), rings are present in each peg, gn(level of the node), hn(heuristic function is called and return value is assigned to hn), fn which is the evaluation function where f(n)=g(n)+h(n) and children nodes. This helped in creation of a new node with less difficulty.
7. Generate_child_nodes function generates all the child nodes which are not visited. Here the child nodes are the possible moves for a particular state.
   For example,
   Node x:
   A:[3,2]
   B:[1]
   C:[]
   Possible child nodes of node x are

Node c1:	Node c2:	Node c3:
A:[3]	A:[3,2,1]	A:[3,2]
B:[1]	B:[]	B:[]
C:[2]	C:[]	C:[1]

8. To get the tracing of A* search algorithm we also have implemented a function which prints current status that is nodes in open and in closed along with evaluation function value f(n). This implementation is in print_open_closed function.

7 A COPY OF THE PROGRAM RUNS

7.1 A* search Algorithm

1. Below result is generated using heuristic_kavitha() function

Anaconda Prompt (anaconda3) - python ./Project2_A05252367_ASTAR.py

```
(base) C:\Users\KAVITHA\OneDrive\Documents\GitHub\AI_Project2>python ./Project2_A05252367_ASTAR.py

starting A* algorithm for 3 disks
Disks are placed in target pole B
g(n): 7  h(n): 0  f(n)= 7
A --> []
B --> [3, 2, 1]
C --> []
total number of nodes generated :  16 Total number of Best nodes expanded :  11
Elapsed time: 0.0010008811950683594 seconds
Memory consumed :  912.203125 MB

starting A* algorithm for 4 disks
Disks are placed in target pole B
g(n): 15  h(n): 0  f(n)= 15
A --> []
B --> [4, 3, 2, 1]
C --> []
total number of nodes generated :  43 Total number of Best nodes expanded :  38
Elapsed time: 0.0009999275207519531 seconds
Memory consumed :  912.2734375 MB

starting A* algorithm for 5 disks
Disks are placed in target pole B
g(n): 31  h(n): 0  f(n)= 31
A --> []
B --> [5, 4, 3, 2, 1]
C --> []
total number of nodes generated :  126 Total number of Best nodes expanded :  118
Elapsed time: 0.2125537395477295 seconds
Memory consumed :  912.3984375 MB

starting A* algorithm for 6 disks
Disks are placed in target pole B
g(n): 63  h(n): 0  f(n)= 63
A --> []
B --> [6, 5, 4, 3, 2, 1]
C --> []
total number of nodes generated :  415 Total number of Best nodes expanded :  401
Elapsed time: 0.008997917175292969 seconds
Memory consumed :  912.79296875 MB

starting A* algorithm for 7 disks
Disks are placed in target pole B
g(n): 129  h(n): 0  f(n)= 129
```

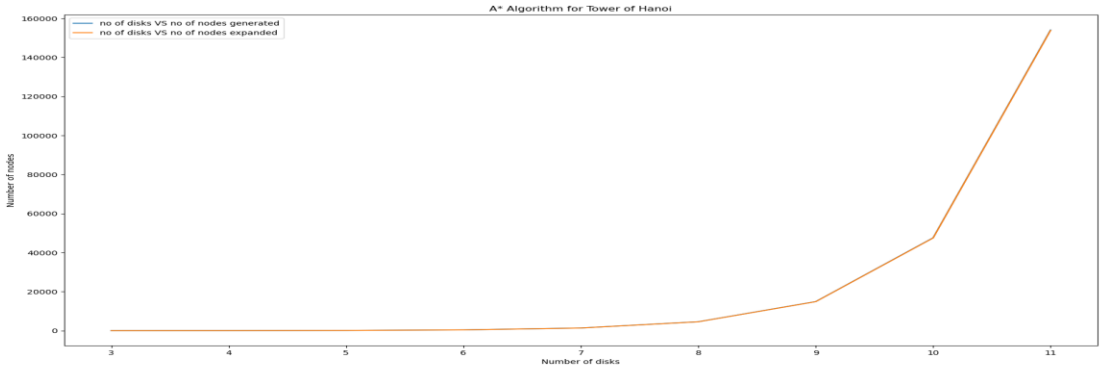
Anaconda Prompt (anaconda3) - python ./Project2_A05252367_ASTAR.py

```
starting A* algorithm for 7 disks
Disks are placed in target pole B
g(n): 129  h(n): 0  f(n)= 129
A --> []
B --> [7, 6, 5, 4, 3, 2, 1]
C --> []
total number of nodes generated :  1375 Total number of Best nodes expanded :  1355
Elapsed time: 0.055626869201660156 seconds
Memory consumed :  914.17578125 MB

starting A* algorithm for 8 disks
Disks are placed in target pole B
g(n): 257  h(n): 0  f(n)= 257
A --> []
B --> [8, 7, 6, 5, 4, 3, 2, 1]
C --> []
total number of nodes generated :  4589 Total number of Best nodes expanded :  4551
Elapsed time: 0.5909514427185059 seconds
Memory consumed :  919.0859375 MB

starting A* algorithm for 9 disks
Disks are placed in target pole B
g(n): 529  h(n): 0  f(n)= 529
A --> []
B --> [9, 8, 7, 6, 5, 4, 3, 2, 1]
C --> []
total number of nodes generated :  14908 Total number of Best nodes expanded :  14858
Elapsed time: 6.487391471862793 seconds
Memory consumed :  935.734375 MB

starting A* algorithm for 10 disks
Disks are placed in target pole B
g(n): 1047  h(n): 0  f(n)= 1047
A --> []
B --> [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
C --> []
total number of nodes generated :  47603 Total number of Best nodes expanded :  47493
Elapsed time: 139.62415599822998 seconds
Memory consumed :  989.16796875 MB
```

```
starting A* algorithm for 11 disks
Disks are placed in target pole B
g(n): 2125  h(n): 0  f(n)= 2125
A --> []
B --> [11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
C --> []
total number of nodes generated :  154090 Total number of Best nodes expanded :  153862
Elapsed time: 1048.9064178466797 seconds
Memory consumed :  1165.1015625 MB
```

Number of Disks	Elapsed Time	Memory Used	Nodes Generated	Nodes Expanded
3	0.0010008811950683594	912.203125	16	11
4	0.0009999275207519531	912.2734375	43	38
5	0.2125537395477295	912.3984375	126	118
6	0.008997917175292969	912.79296875	415	401
7	0.055626869201660156	914.17578125	1375	1355
8	0.5909514427185059	919.0859375	4589	4551
9	6.487391471862793	935.734375	14908	14858
10	139.62415599822998	989.16796875	47603	47493
11	1048.9064178466797	1165.1015625	154090	153862

Plot between number of disks and number of nodes generated



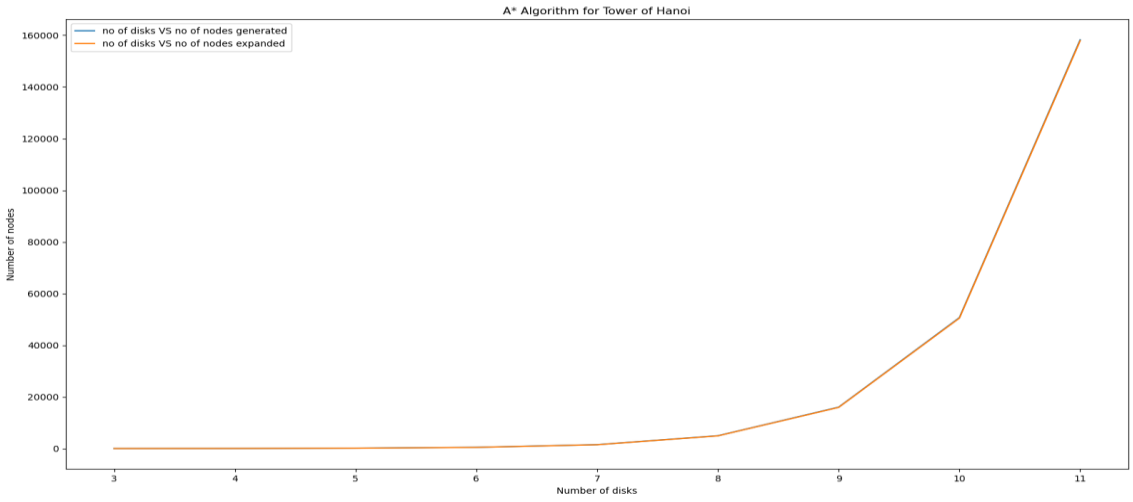
2. Below result is generated using heuristic_sakshi() function

```
(base) C:\Users\KAVITHA\OneDrive\Documents\GitHub\AI_Project2>python ./Project2_A05252367_ASTAR.py

starting A* algorithm for 3 disks
Disks are placed in target pole B
g(n): 7  h(n): 0  f(n)= 7
A --> []
B --> [3, 2, 1]
C --> []
total number of nodes generated :  23 Total number of Best nodes expanded :  17
Elapsed time: 0.0009958744049072266 seconds
Memory consumed :  912.328125 MB

starting A* algorithm for 4 disks
Disks are placed in target pole B
g(n): 15  h(n): 0  f(n)= 15
A --> []
B --> [4, 3, 2, 1]
C --> []
total number of nodes generated :  55 Total number of Best nodes expanded :  47
Elapsed time: 0.10480165481567383 seconds
Memory consumed :  912.390625 MB

starting A* algorithm for 5 disks
Disks are placed in target pole B
g(n): 31  h(n): 0  f(n)= 31
A --> []
B --> [5, 4, 3, 2, 1]
C --> []
total number of nodes generated :  163 Total number of Best nodes expanded :  147
Elapsed time: 0.10861682891845703 seconds
Memory consumed :  912.51171875 MB

starting A* algorithm for 6 disks
Disks are placed in target pole B
g(n): 63  h(n): 0  f(n)= 63
A --> []
B --> [6, 5, 4, 3, 2, 1]
C --> []
total number of nodes generated :  489 Total number of Best nodes expanded :  473
Elapsed time: 0.014106512069702148 seconds
Memory consumed :  913.015625 MB
```

```
Anaconda Prompt (anaconda3) - python ./Project2_A05252367_ASTAR.py
starting A* algorithm for 7 disks
Disks are placed in target pole B
g(n): 127  h(n): 0  f(n)= 127
A --> []
B --> [7, 6, 5, 4, 3, 2, 1]
C --> []
total number of nodes generated :  1519 Total number of Best nodes expanded :  1491
Elapsed time: 0.06883740425109863 seconds
Memory consumed :  914.4765625 MB

starting A* algorithm for 8 disks
Disks are placed in target pole B
g(n): 255  h(n): 0  f(n)= 255
A --> []
B --> [8, 7, 6, 5, 4, 3, 2, 1]
C --> []
total number of nodes generated :  4965 Total number of Best nodes expanded :  4913
Elapsed time: 0.665980339050293 seconds
Memory consumed :  919.67578125 MB

starting A* algorithm for 9 disks
Disks are placed in target pole B
g(n): 511  h(n): 0  f(n)= 511
A --> []
B --> [9, 8, 7, 6, 5, 4, 3, 2, 1]
C --> []
total number of nodes generated :  16031 Total number of Best nodes expanded :  15933
Elapsed time: 7.866077423095703 seconds
Memory consumed :  937.59375 MB

starting A* algorithm for 10 disks
Disks are placed in target pole B
g(n): 1023  h(n): 0  f(n)= 1023
A --> []
B --> [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
C --> []
total number of nodes generated :  50687 Total number of Best nodes expanded :  50499
Elapsed time: 168.94315385818481 seconds
Memory consumed :  994.04296875 MB
```

```
starting A* algorithm for 11 disks
Disks are placed in target pole B
g(n): 2047  h(n): 0  f(n)= 2047
A --> []
B --> [11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
C --> []
total number of nodes generated :  158233 Total number of Best nodes expanded :  157891
Elapsed time: 1090.3040051460266 seconds
Memory consumed :  1171.515625 MB
```

Number of Disks	Elapsed Time	Memory Used	Nodes Generated	Nodes Expanded
3	0.0009958744049072266	912.328125	23	17
4	0.10480165481567383	912.390625	55	47
5	0.10861682891845703	912.51171875	163	147
6	0.014106512069702148	913.015625	489	473
7	0.06883740425109863	914.4765625	1519	1491
8	0.665980339050293	919.67578125	4965	4913
9	7.866077423095703	937.59375	16031	15933
10	168.94315385818481	994.04296875	50687	50499
11	1090.3040051460266	1171.515625	158233	157891



3. Below result is generated using heuristic_Roopika() function

Anaconda Prompt (anaconda3) - python ./Project2_A05252367_ASTAR.py

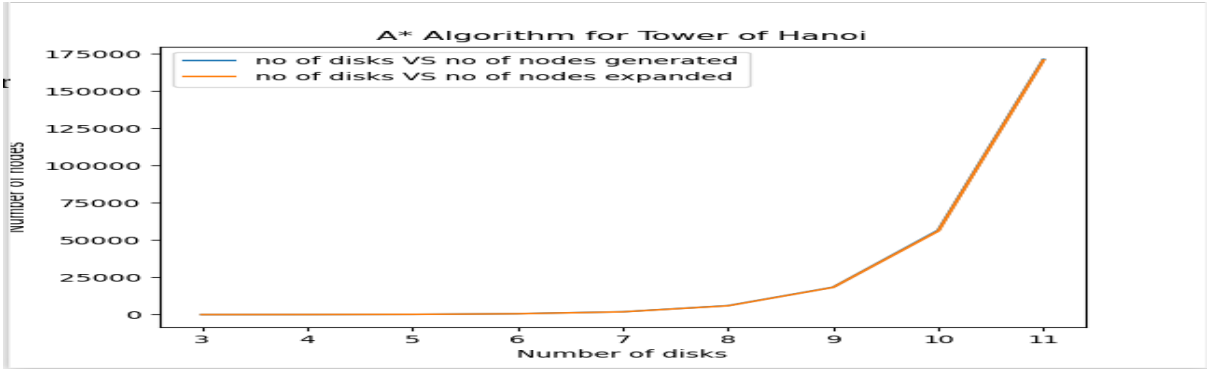
```
(base) C:\Users\KAVITHA\OneDrive\Documents\GitHub\AI_Project2>python ./Project2_A05252367_ASTAR.py

starting A* algorithm for 3 disks
Disks are placed in target pole B
g(n): 7  h(n): 0  f(n)= 7
A --> []
B --> [3, 2, 1]
C --> []
total number of nodes generated :  17 Total number of Best nodes expanded :  14
Elapsed time: 0.0010852813720703125 seconds
Memory consumed :  912.21875 MB

starting A* algorithm for 4 disks
Disks are placed in target pole B
g(n): 15  h(n): 0  f(n)= 15
A --> []
B --> [4, 3, 2, 1]
C --> []
total number of nodes generated :  60 Total number of Best nodes expanded :  52
Elapsed time: 0.10694408416748047 seconds
Memory consumed :  912.28515625 MB

starting A* algorithm for 5 disks
Disks are placed in target pole B
g(n): 31  h(n): 0  f(n)= 31
A --> []
B --> [5, 4, 3, 2, 1]
C --> []
total number of nodes generated :  185 Total number of Best nodes expanded :  176
Elapsed time: 0.12148690223693848 seconds
Memory consumed :  912.4375 MB

starting A* algorithm for 6 disks
Disks are placed in target pole B
g(n): 63  h(n): 0  f(n)= 63
A --> []
B --> [6, 5, 4, 3, 2, 1]
C --> []
total number of nodes generated :  618 Total number of Best nodes expanded :  586
Elapsed time: 0.022615671157836914 seconds
Memory consumed :  913.04296875 MB
```

Anaconda Prompt (anaconda3) - python ./Project2_A05252367_ASTAR.py

```
starting A* algorithm for 7 disks
Disks are placed in target pole B
g(n): 127  h(n): 0  f(n)= 127
A --> []
B --> [7, 6, 5, 4, 3, 2, 1]
C --> []
total number of nodes generated :  1901 Total number of Best nodes expanded :  1868
Elapsed time: 0.08923006057739258 seconds
Memory consumed :  914.921875 MB

starting A* algorithm for 8 disks
Disks are placed in target pole B
g(n): 255  h(n): 0  f(n)= 255
A --> []
B --> [8, 7, 6, 5, 4, 3, 2, 1]
C --> []
total number of nodes generated :  6018 Total number of Best nodes expanded :  5914
Elapsed time: 0.8905470371246338 seconds
Memory consumed :  920.796875 MB

starting A* algorithm for 9 disks
Disks are placed in target pole B
g(n): 511  h(n): 0  f(n)= 511
A --> []
B --> [9, 8, 7, 6, 5, 4, 3, 2, 1]
C --> []
total number of nodes generated :  18389 Total number of Best nodes expanded :  18284
Elapsed time: 10.874289751052856 seconds
Memory consumed :  940.78515625 MB

starting A* algorithm for 10 disks
Disks are placed in target pole B
g(n): 1023  h(n): 0  f(n)= 1023
A --> []
B --> [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
C --> []
total number of nodes generated :  56634 Total number of Best nodes expanded :  56146
Elapsed time: 255.50373888015747 seconds
Memory consumed :  1003.5 MB
```

```
starting A* algorithm for 11 disks
Disks are placed in target pole B
g(n): 2047  h(n): 0  f(n)= 2047
A --> []
B --> [11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
C --> []
total number of nodes generated :  171341 Total number of Best nodes expanded :  170852
Elapsed time: 1564.8071093559265 seconds
Memory consumed :  1190.6875 MB
```

Number of Disks	Elapsed Time	Memory Used	Nodes Generated	Nodes Expanded
3	0.0010852813720703125	912.21875	17	14
4	0.10694408416748047	912.28515625	60	52
5	0.12148690223693848	912.4375	185	176
6	0.022615671157836914	913.04296875	618	586
7	0.08923006057739258	914.921875	1901	1868
8	0.8905470371246338	920.796875	6018	5914
9	10.874289751052856	940.78515625	18389	18284
10	255.50373888015747	1003.5	56634	56146
11	1564.8071093559265	1190.6875	171341	170852



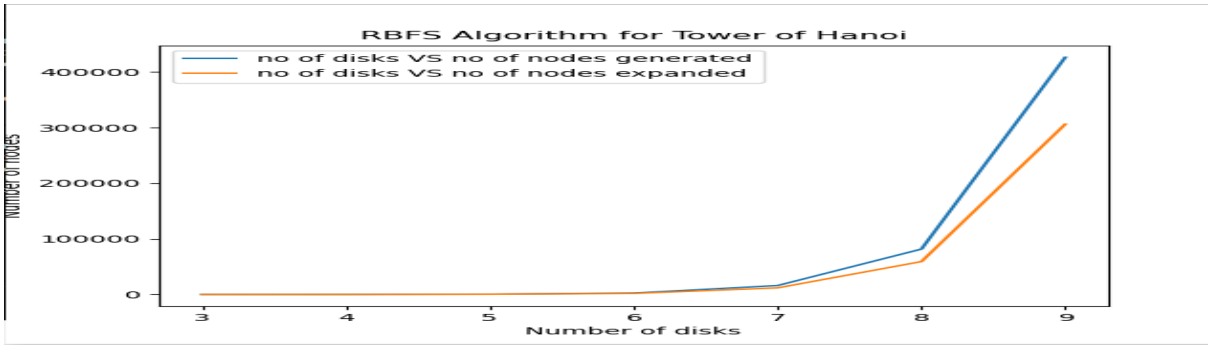
7.2 Recursive Best First Search

1. Below result is generated using heuristic_kavitha() function

```
Anaconda Prompt (anaconda3) - python ./Project2_A05252367_RBFS.py
(base) C:\Users\KAVITHA\OneDrive\Documents\GitHub\AI_Project2>python ./Project2_A05252367_RBFS.py

starting Recursive best first search algorithm for 3 disks
g(n): 9  h(n): 0  f(n)= 10
A --> []
B --> [3, 2, 1]
C --> []
total number of nodes generated :  18 Total number of Best nodes expanded :  11
Elapsed time: 0.0012586116790771484 seconds
Memory consumed :  51.515625 MB

starting Recursive best first search algorithm for 4 disks
g(n): 15  h(n): 0  f(n)= 20
A --> []
B --> [4, 3, 2, 1]
C --> []
total number of nodes generated :  77 Total number of Best nodes expanded :  62
Elapsed time: 0.0020341873168945312 seconds
Memory consumed :  51.515625 MB

starting Recursive best first search algorithm for 5 disks
g(n): 41  h(n): 0  f(n)= 42
A --> []
B --> [5, 4, 3, 2, 1]
C --> []
total number of nodes generated :  572 Total number of Best nodes expanded :  432
Elapsed time: 0.005985736846923828 seconds
Memory consumed :  51.59765625 MB

starting Recursive best first search algorithm for 6 disks
g(n): 69  h(n): 0  f(n)= 74
A --> []
B --> [6, 5, 4, 3, 2, 1]
C --> []
total number of nodes generated :  2519 Total number of Best nodes expanded :  1886
Elapsed time: 0.042530059814453125 seconds
Memory consumed :  51.97265625 MB
```

Anaconda Prompt (anaconda3) - python ./Project2_A05252367_RBFS.py

```
starting Recursive best first search algorithm for 7 disks
g(n): 147  h(n): 0  f(n)= 148
A --> []
B --> [7, 6, 5, 4, 3, 2, 1]
C --> []
total number of nodes generated :  16046 Total number of Best nodes expanded :  11839
Elapsed time: 0.6967487335205078 seconds
Memory consumed :  54.265625 MB

starting Recursive best first search algorithm for 8 disks
g(n): 285  h(n): 0  f(n)= 290
A --> []
B --> [8, 7, 6, 5, 4, 3, 2, 1]
C --> []
total number of nodes generated :  81602 Total number of Best nodes expanded :  59121
Elapsed time: 14.777370452880086 seconds
Memory consumed :  62.59375 MB

starting Recursive best first search algorithm for 9 disks
g(n): 589  h(n): 0  f(n)= 590
A --> []
B --> [9, 8, 7, 6, 5, 4, 3, 2, 1]
C --> []
total number of nodes generated :  426710 Total number of Best nodes expanded :  306323
Elapsed time: 500.501957654953 seconds
Memory consumed :  95.19140625 MB
---------------  ----------------------  ------------  ----------------  ---------------
Number of Disks  Elapsed Time           Memory Used   Nodes Generated   Nodes Expanded
3                0.0012586116790771484  51.515625     18                11
4                0.0020341873168945312  51.515625     77                62
5                0.005985736846923828   51.59765625   572               432
6                0.04253005981445312    51.97265625   2519              1886
7                0.6967487335205078     54.265625     16046             11839
8                14.777370452880086     62.59375      81602             59121
9                500.501957654953       95.19140625   426710            306323
---------------  ----------------------  ------------  ----------------  ---------------
```

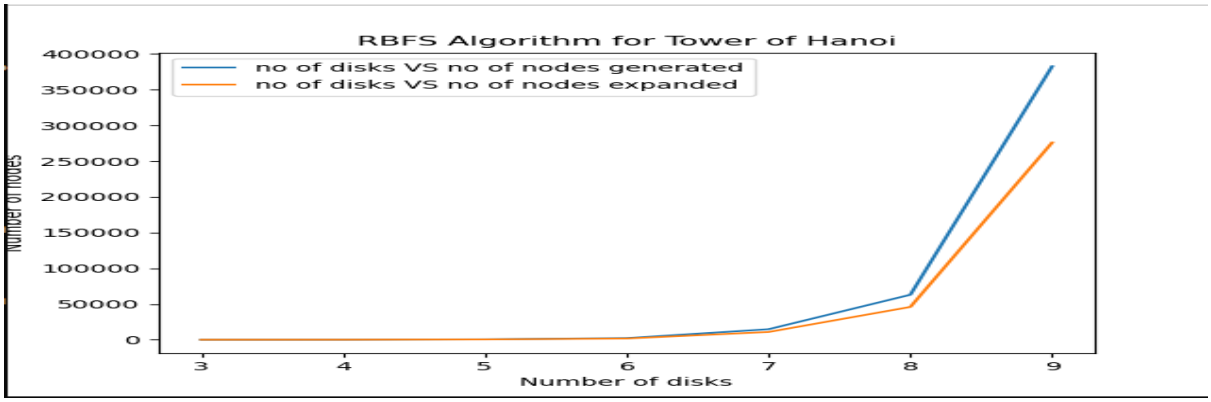


2. Below result is generated using heuristic_sakshi() function

Anaconda Prompt (anaconda3) - python ./Project2_A05252367_RBFS.py

```
(base) C:\Users\KAVITHA\OneDrive\Documents\GitHub\AI_Project2>python ./Project2_A05252367_RBFS.py

starting Recursive best first search algorithm for 3 disks
g(n): 7  h(n): 0  f(n)= 9
A --> []
B --> [3, 2, 1]
C --> []
total number of nodes generated :  25 Total number of Best nodes expanded :  18
Elapsed time: 0.0009279251098632812 seconds
Memory consumed :  51.32421875 MB

starting Recursive best first search algorithm for 4 disks
g(n): 15  h(n): 0  f(n)= 17
A --> []
B --> [4, 3, 2, 1]
C --> []
total number of nodes generated :  87 Total number of Best nodes expanded :  65
Elapsed time: 0.0010714530944824219 seconds
Memory consumed :  51.32421875 MB

starting Recursive best first search algorithm for 5 disks
g(n): 33  h(n): 0  f(n)= 35
A --> []
B --> [5, 4, 3, 2, 1]
C --> []
total number of nodes generated :  580 Total number of Best nodes expanded :  447
Elapsed time: 0.010275602340698242 seconds
Memory consumed :  51.47265625 MB

starting Recursive best first search algorithm for 6 disks
g(n): 63  h(n): 0  f(n)= 65
A --> []
B --> [6, 5, 4, 3, 2, 1]
C --> []
total number of nodes generated :  2253 Total number of Best nodes expanded :  1705
Elapsed time: 0.04609370231628418 seconds
Memory consumed :  51.9296875 MB
```

```
starting Recursive best first search algorithm for 7 disks
g(n): 135  h(n): 0  f(n)= 137
A --> []
B --> [7, 6, 5, 4, 3, 2, 1]
C --> []
total number of nodes generated :  14691 Total number of Best nodes expanded :  10851
Elapsed time: 0.7517802715301514 seconds
Memory consumed :  53.84375 MB

starting Recursive best first search algorithm for 8 disks
g(n): 255  h(n): 0  f(n)= 257
A --> []
B --> [8, 7, 6, 5, 4, 3, 2, 1]
C --> []
total number of nodes generated :  62965 Total number of Best nodes expanded :  45886
Elapsed time: 10.766086101531982 seconds
Memory consumed :  60.61328125 MB

starting Recursive best first search algorithm for 9 disks
g(n): 539  h(n): 0  f(n)= 541
A --> []
B --> [9, 8, 7, 6, 5, 4, 3, 2, 1]
C --> []
total number of nodes generated :  382930 Total number of Best nodes expanded :  276157
Elapsed time: 663.6565413475037 seconds
Memory consumed :  92.6953125 MB
---------------  ---------------------  -----------  ---------------  --------------
Number of Disks  Elapsed Time           Memory Used  Nodes Generated  Nodes Expanded
3                0.0009279251098632812  51.32421875  25               18
4                0.0010714530944824219  51.32421875  87               65
5                0.010275602340698242   51.47265625  580              447
6                0.04609370231628418    51.9296875   2253             1705
7                0.7517802715301514     53.84375     14691            10851
8                10.766086101531982     60.61328125  62965            45886
9                663.6565413475037      92.6953125   382930           276157
---------------  ---------------------  -----------  ---------------  --------------
```

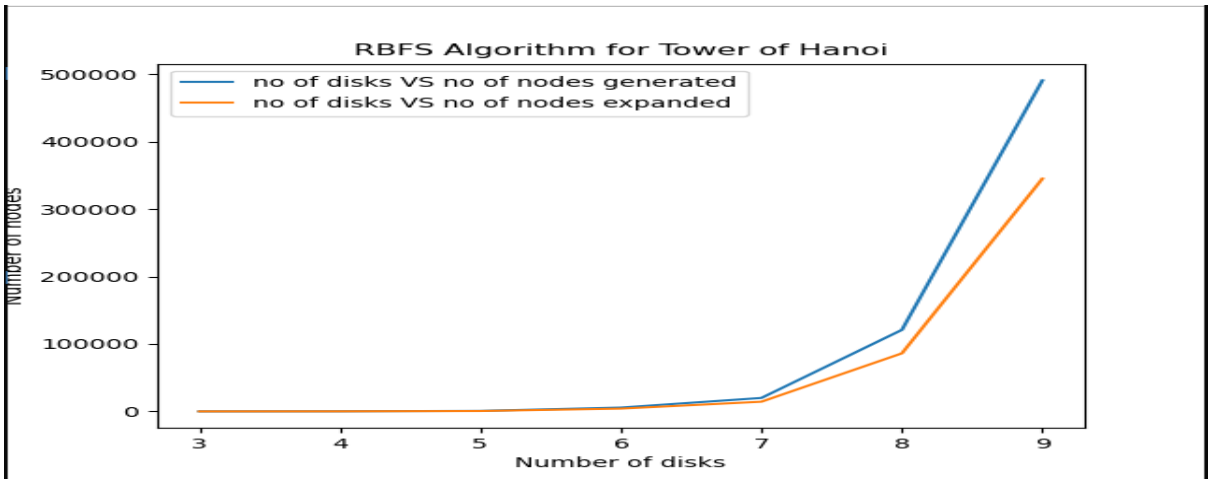


3. Below result is generated using heuristic_roopika() function

Anaconda Prompt (anaconda3) - python ./Project2_A05252367_RBFS.py

```
(base) C:\Users\KAVITHA\OneDrive\Documents\GitHub\AI_Project2>python ./Project2_A05252367_RBFS.py

starting Recursive best first search algorithm for 3 disks
g(n): 7  h(n): 0  f(n)= 7
A --> []
B --> [3, 2, 1]
C --> []
total number of nodes generated :  25 Total number of Best nodes expanded :  19
Elapsed time: 0.0011761188507080078 seconds
Memory consumed :  51.53515625 MB

starting Recursive best first search algorithm for 4 disks
g(n): 15  h(n): 0  f(n)= 15
A --> []
B --> [4, 3, 2, 1]
C --> []
total number of nodes generated :  184 Total number of Best nodes expanded :  135
Elapsed time: 0.002994537353515625 seconds
Memory consumed :  51.53515625 MB

starting Recursive best first search algorithm for 5 disks
g(n): 31  h(n): 0  f(n)= 31
A --> []
B --> [5, 4, 3, 2, 1]
C --> []
total number of nodes generated :  971 Total number of Best nodes expanded :  705
Elapsed time: 0.009000062942504883 seconds
Memory consumed :  51.71875 MB

starting Recursive best first search algorithm for 6 disks
g(n): 63  h(n): 0  f(n)= 63
A --> []
B --> [6, 5, 4, 3, 2, 1]
C --> []
total number of nodes generated :  5320 Total number of Best nodes expanded :  3762
Elapsed time: 0.09618139266967773 seconds
Memory consumed :  52.59765625 MB
```

```
starting Recursive best first search algorithm for 7 disks
g(n): 127  h(n): 0  f(n)= 127
A --> []
B --> [7, 6, 5, 4, 3, 2, 1]
C --> []
total number of nodes generated :  24822 Total number of Best nodes expanded :  17303
Elapsed time: 1.3827438354492188 seconds
Memory consumed :  55.19140625 MB

starting Recursive best first search algorithm for 8 disks
g(n): 255  h(n): 0  f(n)= 255
A --> []
B --> [8, 7, 6, 5, 4, 3, 2, 1]
C --> []
total number of nodes generated :  113788 Total number of Best nodes expanded :  78318
Elapsed time: 25.759192943572998 seconds
Memory consumed :  64.33984375 MB

starting Recursive best first search algorithm for 9 disks
g(n): 511  h(n): 0  f(n)= 511
A --> []
B --> [9, 8, 7, 6, 5, 4, 3, 2, 1]
C --> []
total number of nodes generated :  489793 Total number of Best nodes expanded :  334328
Elapsed time: 490.03018260002136 seconds
Memory consumed :  101.71484375 MB
---------------  ---------------------  ------------  ---------------  --------------
Number of Disks  Elapsed Time           Memory Used   Nodes Generated  Nodes Expanded
3                0.0011761188507080078  51.53515625   25               19
4                0.002994537353515625   51.53515625   184              135
5                0.0090000629425048833  51.71875      971              705
6                0.09618139266967773    52.59765625   5320             3762
7                1.3827438354492188     55.19140625   24822            17303
8                25.759192943572998     64.33984375   113788           78318
9                490.03018260002136     101.71484375  489793           334328
---------------  ---------------------  ------------  ---------------  --------------
```



8 TABULATION OF RESULTS

8.1 A* Search Algorithm results:

Heuristic_function	Numer of disks(n)	Numer of nodes generated	Number of nodes expanded	Memory utilized	Execution Time
heuristic_kavitha	3	16	11	912.203125	0.001000881195
	4	43	38	912.2734375	0.000999927520
	5	126	118	912.3984375	0.212553739547
	6	415	401	912.79296875	0.008997917175
	7	1375	1355	914.17578125	0.055626869201
	8	4589	4551	919.0859375	0.590951442718
	9	14908	14858	935.734375	6.487391471862
	10	47603	47493	989.16796875	139.6241559982
	11	154090	153862	1165.1015625	1048.906417846

heuristic_sakshi	3	23	17	912.328125	0.000995874404
	4	55	47	912.390625	0.104801654815
	5	163	147	912.51171875	0.108616828918
	6	489	473	913.015625	0.014106512069
	7	1519	1491	914.4765625	0.068837404251
	8	4965	4913	919.67578125	0.665980339050
	9	16031	15933	937.59375	7.866077423095
	10	50687	50499	994.04296875	168.9431538581
	11	158233	157891	1171.515625	1090.304005146
heuristic_roopika	3	17	14	912.21875	0.0010852813720
	4	60	52	912.28515625	0.1069440841674
	5	185	176	912.4375	0.1214869022369
	6	618	586	913.04296875	0.0226156711578
	7	1901	1868	914.921875	0.0892300605773
	8	6018	5914	920.796875	0.8905470371246
	9	18389	18284	940.78515625	10.874289751052
	10	56634	56146	1003.5	255.50373888015
	11	171341	170852	1190.6875	1564.8071093559

From the above results, we can clearly say that A* search best algorithm as the number of nodes generated and expanded are less when compared to RBFS. However, we also need to consider that RBFS is consuming very less memory compared to A*.

The overall time complexity of the A* search algorithm for solving the Tower of Hanoi problem is expected to be exponential, $O(3^n)$, where n is the number of disks.

The RBFS time complexity is $O(b^{m/w})$, where m is the maximum depth of the recursion, and w is the effective branching factor.

8.2 Recursive Best First Search Algorithm results:

Heuristic_function	**Number of disks**	**Number of nodes generated**	**Number of nodes expanded**	**Memory used**	**Execution time**
heuristic_kavitha	3	18	11	51.515625	0.0012586116790
	4	77	62	51.515625	0.0020341873168
	5	572	432	51.59765625	0.0059857368469
	6	2519	1886	51.97265625	0.0425300598144
	7	16046	11839	54.265625	0.6967487335205
	8	81602	59121	62.59375	14.777370452880
	9	426710	306323	95.19140625	500.50195765495
heuristic_sakshi	3	25	18	51.32421875	0.0009279251098
	4	87	65	51.32421875	0.0010714530944
	5	580	447	51.47265625	0.0102756023406
	6	2253	1705	51.9296875	0.0460937023162
	7	14691	10851	53.84375	0.7517802715301
	8	62965	45886	60.61328125	10.766086101531
	9	382930	276157	92.6953125	389.65654134750
heuristic_roopika	3	25	19	51.53515625	0.0011761188507
	4	184	135	51.53515625	0.0029945373535

	5	971	705	51.71875	0.0090000629425
	6	5320	3762	52.59765625	0.0961813926696
	7	24822	17303	55.19140625	1.3827438354492
	8	113788	78318	64.33984375	25.759192943572
	9	489793	334328	101.71484375	490.03018260002

9 ANALYSIS OF THE RESULTS

9.1 Analysis of A* search Algorithm results

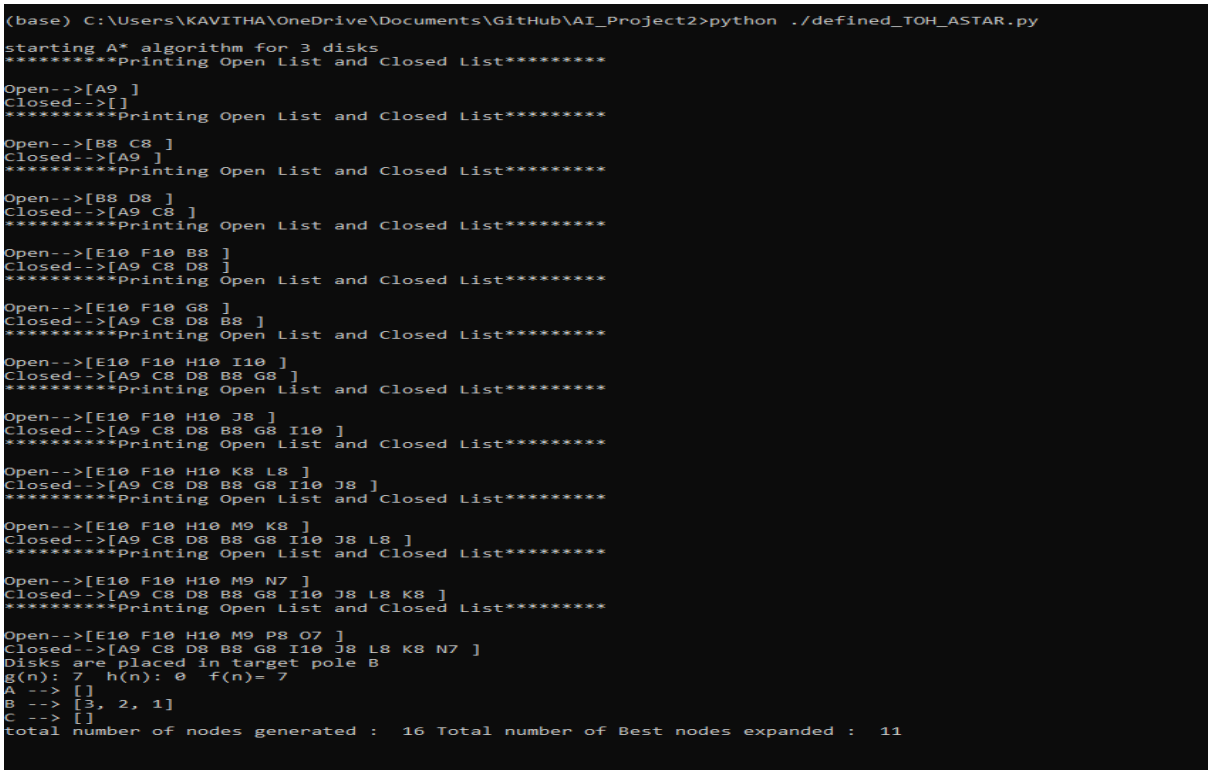
Below are Analysis of the results:

1.check_in_visited function in both algorithms have improved the execution time and lessened the number of nodes generated. The below screenshot is the execution time and nodes generated for 3 disks by the A* and RBFS algorithms.

```
starting A* algorithm for 3 disks
Disks are placed in target pole B
g(n): 7  h(n): 0  f(n)= 7
A --> []
B --> [3, 2, 1]
C --> []
Nodes in visited_nodes list which is updated by check_in_visited() function:
 ['A321BC', 'A32B1C', 'A32BC1', 'A3B2C1', 'A31B2C', 'A3B21C', 'A3B1C2', 'A31BC2', 'A3BC21', 'AB3C21', 'A1B3C2', 'AB31C2', 'A2B31C', 'A1B32C', 'AB321C', 'AB32C1']
total number of nodes generated :  16 Total number of Best nodes expanded :  11
Elapsed time: 0.007376194000244141 seconds
Memory consumed :  912.20703125 MB
--------------  -------------------  ------------  --------------  --------------
Number of Disks  Elapsed Time        Memory Used   Nodes Generated  Nodes Expanded
3               0.007376194000244141  912.20703125  16              11
--------------  -------------------  ------------  --------------  --------------
```

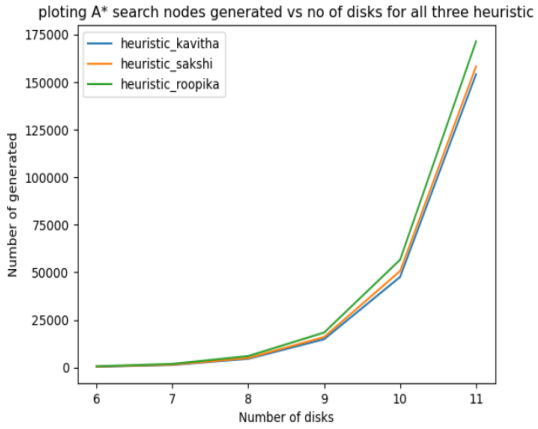
For 3 disks there are only 16 nodes generated and elapsed time is also very less i.e, 0.007376194. Visited_nodes list in above screenshot [‘A321BC’,’A32B1C’, ……]. Here, A32B1C represents peg A contains rings 3,2 and peg B contains the ring 1. Rings 3,2,1 are the sizes of the rings.

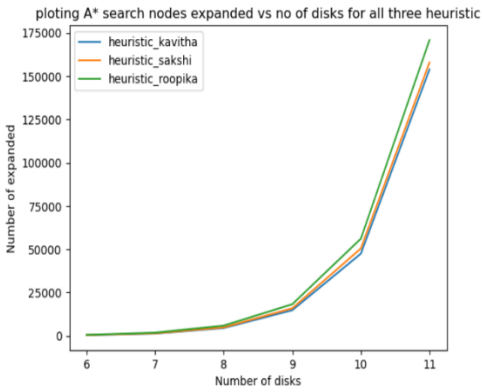
2. By choosing open data structure as stack it helped to access the list easily and efficiently pop the best node from stack. Below is the tracing of A* algorithm which shows open and close status for each run.



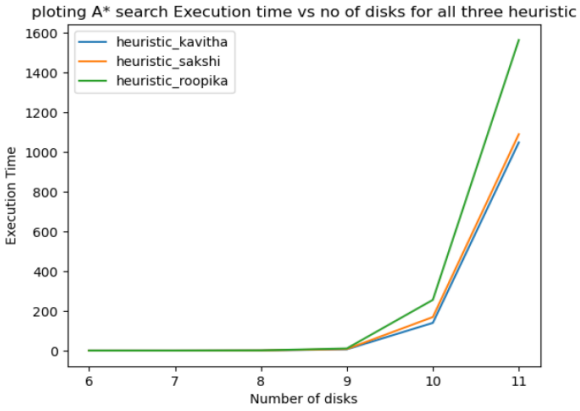
In the above screenshot, the first iteration contains the initial node [A9] in open where A is the state name and 9 is the f(n) value. A9 is the best node in the open so it is popped and the child nodes of A9 are [B8,C8] added. This approach improves readability of algorithms.

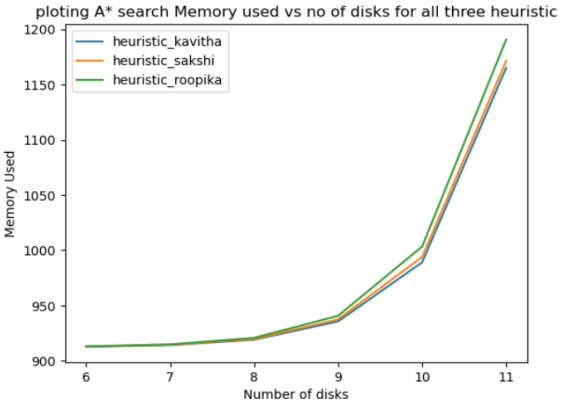
3. By analyzing the results of all heuristic functions. I conclude that for A* algorithm the heuristic_kavitha function has given the best output as this heuristic considers both position and size of rings present in all the 3 pegs. The execution time, memory utilized, number of nodes generated and expanded are less in heuristic_kavitha function when compared with heuristic_sakshi, heuristic_roopika. The analysis of these parameters can be done by considering 6 to 11 disks and all three heuristic function results. The results are plotted as below-





In the above nodes generated and nodes expanded graph shows that blue line(heuristc_kavitha) is having less count.





In the above execution time and memory used graph shows that blue line (heuristic_kavitha) is having time and memory used compared to other heuristic functions.

Below are the execution results of each heuristic function for 9 disks.
heuristic_kavitha()

```
starting A* algorithm for 9 disks
Disks are placed in target pole B
g(n): 529  h(n): 0  f(n)= 529
A --> []
B --> [9, 8, 7, 6, 5, 4, 3, 2, 1]
C --> []
total number of nodes generated :  14908 Total number of Best nodes expanded :  14858
Elapsed time: 6.470529317855835 seconds
Memory consumed :  935.328125 MB
--------------  -----------------  -----------  ---------------  --------------
Number of Disks  Elapsed Time       Memory Used  Nodes Generated  Nodes Expanded
9                6.470529317855835  935.328125   14908            14858
--------------  -----------------  -----------  ---------------  --------------
```

heuristic_roopika()

```
starting A* algorithm for 9 disks
Disks are placed in target pole B
g(n): 511  h(n): 0  f(n)= 511
A --> []
B --> [9, 8, 7, 6, 5, 4, 3, 2, 1]
C --> []
total number of nodes generated :  18389 Total number of Best nodes expanded :  18284
Elapsed time: 10.492246150970459 seconds
Memory consumed :  941.1484375 MB
--------------  -----------------  -----------  --------------  -------------
Number of Disks  Elapsed Time       Memory Used  Nodes Generated  Nodes Expanded
9                10.492246150970459  941.1484375  18389           18284
--------------  -----------------  -----------  --------------  -------------
```

heuristic_sakshi()

```
starting A* algorithm for 9 disks
Disks are placed in target pole B
g(n): 511  h(n): 0  f(n)= 511
A --> []
B --> [9, 8, 7, 6, 5, 4, 3, 2, 1]
C --> []
total number of nodes generated :  16031 Total number of Best nodes expanded :  15933
Elapsed time: 7.599240779876709 seconds
Memory consumed :  937.39453125 MB
--------------  -----------------  -----------  --------------  -------------
Number of Disks  Elapsed Time       Memory Used  Nodes Generated  Nodes Expanded
9                7.599240779876709  937.39453125  16031           15933
--------------  -----------------  -----------  --------------  -------------
```

Clearly from 3 outputs of heuristic functions, heuristic_kavitha output is most efficient and a lesser number of nodes are generated compared to other two heuristic functions.

4. Print_tower_of_hanoi function is used to improve readability. By calling this function with any node it displays the rings present in each peg. Generate_child_node function is used to generate the child nodes for a particular node.

   Below is the execution results to understand how print_tower_of_hanoi and generate_child_node function works-

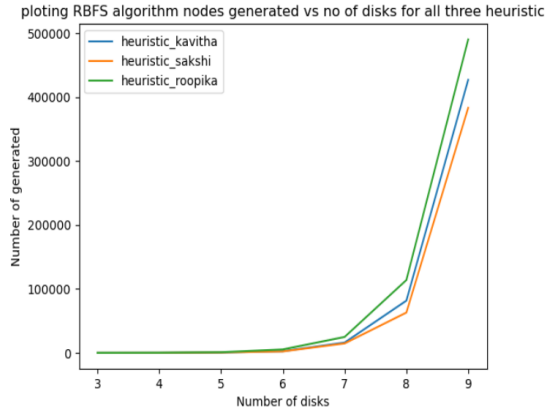
```
starting A* algorithm for 3 disks
*****************Best Node*****************
g(n): 0  h(n): 9  f(n)= 9
A --> [3, 2, 1]
B --> []
C --> []
***child nodes of Best node are***
g(n): 1  h(n): 7  f(n)= 8
A --> [3, 2]
B --> [1]
C --> []
g(n): 1  h(n): 7  f(n)= 8
A --> [3, 2]
B --> []
C --> [1]
```

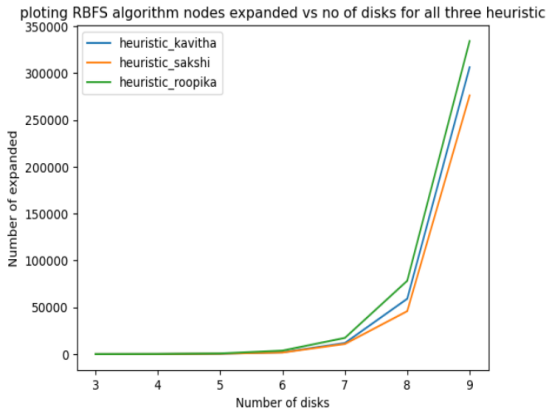
In above result it is shown that, print_tower_of_hanoi function displays the status of the best node which means what rings(3,2,1) present in each peg(A,B,C) and the child nodes of this

best node is generated using generate_child_node function.so, best node has two child nodes which is shown above screenshot.

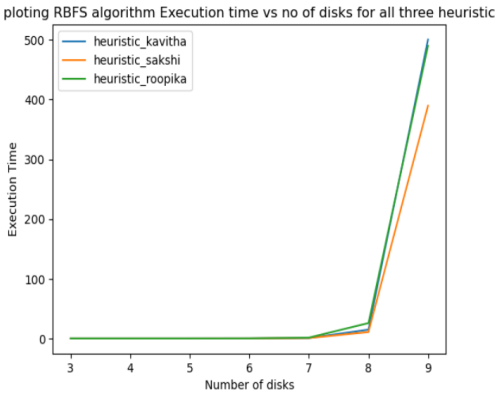
9.2 Analysis of RBFS algorithm results

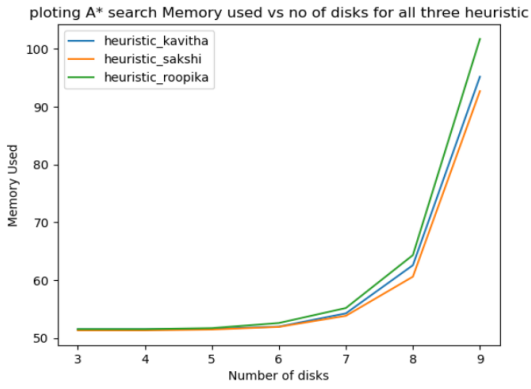
By analyzing the results of all heuristic functions in the RBFS algorithm. I conclude that for RBFS algorithm heuristic_sakshi function has given best output.The execution time, memory utilized, number of nodes generated and expanded are less while heuristic_sakshi function when compared with heuristic_kavitha, heuristic_roopika. The analysis of these parameters can be done by considering 6 to 11 disks and all three heuristic function results. The results are plotted as below-





In the above nodes generated and nodes expanded graph shows that the blue line(heuristc_sakshi) is having less count.





In the above execution time and memory used graph shows that blue line (heuristic_sakshi) is having time and memory used compared to other heuristic functions.

Below are the execution results of each heuristic function for 8 disks.

heuristic_kavitha()

```
starting Recursive best first search algorithm for 8 disks
g(n): 285  h(n): 0  f(n)= 290
A --> []
B --> [8, 7, 6, 5, 4, 3, 2, 1]
C --> []
total number of nodes generated :  81602 Total number of Best nodes expanded :  59121
Elapsed time: 14.039472341537476 seconds
Memory consumed :  62.12109375 MB
--------------  ------------------  -----------  ---------------  --------------
Number of Disks  Elapsed Time        Memory Used  Nodes Generated  Nodes Expanded
8                14.039472341537476  62.12109375  81602            59121
--------------  ------------------  -----------  ---------------  --------------
```

heuristic_roopika()

```
starting Recursive best first search algorithm for 8 disks
g(n): 255  h(n): 0  f(n)= 255
A --> []
B --> [8, 7, 6, 5, 4, 3, 2, 1]
C --> []
total number of nodes generated :  113788 Total number of Best nodes expanded :  78318
Elapsed time: 26.15339732170105 seconds
Memory consumed :  64.0234375 MB
--------------  -----------------  -----------  ---------------  --------------
Number of Disks  Elapsed Time       Memory Used  Nodes Generated  Nodes Expanded
8                26.15339732170105  64.0234375   113788           78318
--------------  -----------------  -----------  ---------------  --------------
```

heuristic_sakshi()

```
starting Recursive best first search algorithm for 8 disks
g(n): 255  h(n): 0  f(n)= 257
A --> []
B --> [8, 7, 6, 5, 4, 3, 2, 1]
C --> []
total number of nodes generated :  62965 Total number of Best nodes expanded :  45886
Elapsed time: 9.05571961402893 seconds
Memory consumed :  60.0078125 MB
--------------  ----------------  -----------  ---------------  --------------
Number of Disks  Elapsed Time      Memory Used  Nodes Generated  Nodes Expanded
8                9.05571961402893  60.0078125   62965            45886
--------------  ----------------  -----------  ---------------  --------------
```

Clearly from 3 outputs of heuristic functions, heuristic_sakshi and heuristic_kavitha output is most efficient in time & memory and a lesser number of nodes are generated compared to other heuristic functions.

10 CONCLUSION

To sum up, the Tower of Hanoi issue can have optimal solutions found using A* as the heuristic is admissible which will never overestimate the cost. By cleverly navigating the state space, they ensure the quickest route to the desired state. In contrast, RBFS uses less RAM due to its recursive nature. The heuristic function's quality determines how effective both algorithms are. The cost to attain the objective state can be precisely estimated using a well-designed heuristic function. The heuristic_kavitha function is the best heuristic function that considers both position and size of rings because, in comparison to the other

two heuristic functions, it required significantly less time and memory and produced and expanded fewer nodes.

Based on the results, I believe that the A* search algorithm is the most effective way to answer the Tower Hanoi problem. This is because, in just 30 minutes, the A* search strategy can solve the 3 to 11 disk count, but the RBFS approach can only handle the 3 to 9 disk count.

Although RBFS uses less memory than the A* Algorithm since it regenerates the nodes, it is still less efficient.

11 TEAM MEMBER'S CONTRIBUTIONS

11.1 Kavitha Lodagala

- Took the initiative to improve the code, and the project has profited from this proactive approach.
- Had a significant impact in resolving issues or getting past roadblocks in the project.
- Tested the code on a variety of disk counts. In addition to these tested negative scenarios.
- Efficiently shared thoughts, information, and problems with the group.
- Actively participated in the success of the team by working well with the team.
- Finishing projects earlier than expected and finding solutions to difficult issues.
- Team members were assisted and expertise was shared.
- Took more responsibility for additional work to be completed by the project's deadline.
- worked on the Recursive Best First Search (RBFS) algorithm as well as the A* algorithm.
- Analyzed which algorithm will be the best algorithm for solving the Tower of Hanoi Problem.
- Took initiative to implement the algorithm and analyzed the algorithm how to improve by involving some additional features.
- Examined each of the three heuristic functions' efficiency.

11.2 Roopika Ganesh

- Actively participated in the success of the team by working well with others.
- Organized meetings and set the deadlines
- Clearly communicated and interacted with the team members in an effective manner.
- Worked on the Recursive Best First Search (RBFS) algorithm as well as the A* algorithm.
- Tested each scenario's code separately.

11.3 Sakshi Tripathi

- Actively participated in the success of the team by working well with others.
- Helped teammates and imparted knowledge.

- Worked on the Recursive Best First Search (RBFS) algorithm as well as the A* algorithm.
- Collaborated with the team and did the initial analysis.
- Tested the code for all the scenarios. In addition to these tested negative scenarios.
- Actively contributed to the team's success through effective collaboration.

12 REFRENCES

[1] https://en.wikipedia.org/wiki/Tower_of_Hanoi

[2] https://www.w3schools.com/python/python_lists_comprehension.asp

[3] https://www.geeksforgeeks.org/sort-in-python/

[4] https://en.wikipedia.org/wiki/A*_search_algorithm

[5] https://www.iosrjournals.org/iosr-jce/papers/Vol10-issue5/R0105105110.pdf

[6] https://docs.python.org/3/library/time.html

[7] https://www.geeksforgeeks.org/psutil-module-in-python/