CSE5AIF

Artificial Intelligence Fundamentals

Lab 5: Towers of Hanoi with DFS

Lab Objectives

The objective of this lab is to familiarize yourself with the Towers of Hanoi problem ahead of completing the assignment on the same problem.

By the end of this lab you should understand what the Towers of Hanoi problem entails, how to properly represent it, and how to implement a Depth First Search algorithm to solve it.

Background Information

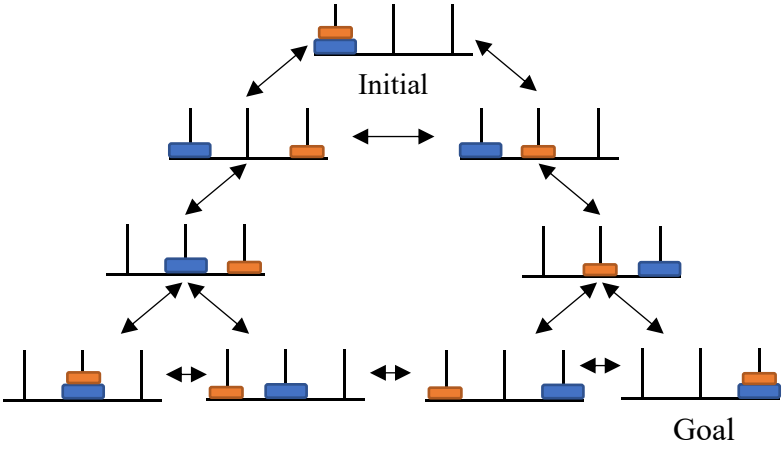
As in the brief for assignment 1, the Towers of Hanoi problem is stated as follows:

> *In a monastery in the deepest parts of Tibet there are three crystal columns and 64 golden rings. The rings are different sizes and rest over the columns. At the beginning of time, all of the rings rested on the leftmost column, and since then the monks have toiled ceaselessly trying to perfectly transfer the rings to their resting place on the final column.*

The objective of this problem is to move the entire stack of rings from the first, to the last column, while obeying 3 simple rules:

1. Only one ring can be moved at time.
2. Each move consists of taking the upper ring from one of the stacks and placing it on the top of another stack or an empty pole.
3. A larger ring must not be placed on top of a smaller ring.

A state space representation of a 3 column, 2 ring, Towers of Hanoi problem is shown in the graph:



Representation

For this lab we will consider a Towers of Hanoi problem with 3 columns and 6 rings/discs; This is displayed in the following figure:

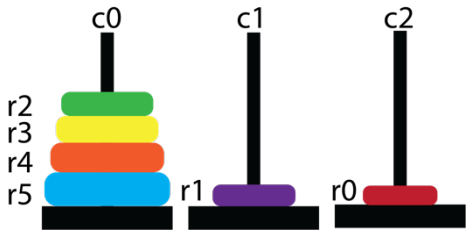
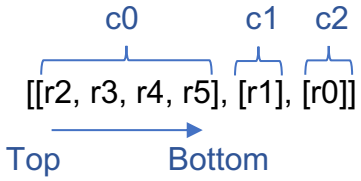


Figure 1.

In order to represent this efficiently, we will utilize a 2-dimensional list. In this list, each index will represent a column where the rings in this column are ordered from top down (i.e. smallest ring -> largest ring). Utilizing this approach will later allow us to easily move a ring from one column to another by popping the top ring off of the list.



Expanding Nodes

To expand a node in the 8-Puzzle problem we had to check that the move was valid by checking all 4 tiles adjacent to our zero tile, as pictured:

1	4	3
7	0	6
5	8	2

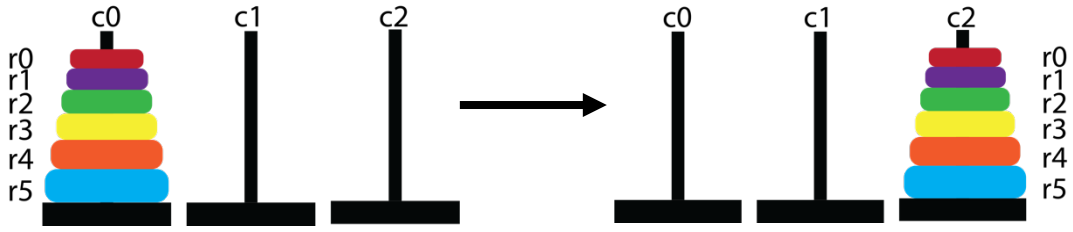
Similarly, expanding a node in the towers of Hanoi problem, requires us to check every possible move for the current node state. To do this we attempt to move the top ring of each column to the other two columns. For example, expanding the node pictured above in **Figure 1** would require checking the following moves:

- Move Ring from Column **0** to Column **1**
- Move Ring from Column **0** to Column **2**
- Move Ring from Column **1** to Column **0**
- Move Ring from Column **1** to Column **2**
- Move Ring from Column **2** to Column **0**
- Move Ring from Column **2** to Column **1**

As evident from the Towers of Hanoi ruleset, many of these moves would not be valid as they would attempt to stack larger rings on top of smaller ring. This is something that must be checked for when completing these moves in code.

Objective State

The objective state for this problem is the node at which all rings are stacked in order from largest to smallest on the final column:

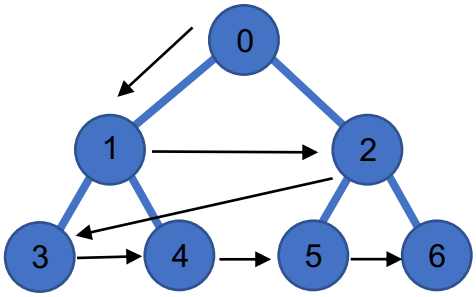


Initial State Objective State

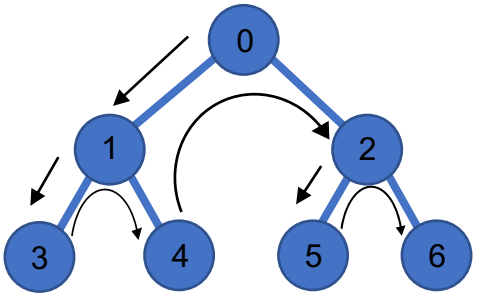
Our elected representation makes checking the objective state very simple. All that is required is a loop from 0 to 5 checking the last stack from top down.

Depth First Search

The following diagrams show the difference between Breadth First Search (BFS) and Depth First Search (DFS).



Breadth First Search



Depth First Search

In previous labs you have implemented the Breadth First Search algorithm using the following pseudocode:

1. We take the node at the front of the **open** list as our **current_node** (initially this is our root node).
2. We check if the **current_node** is our goal node,
    a. if it is, we return the path taken up to this point as the solution;
    b. if it isn't, we continue
3. We now must expand the **current_node** to reveal its neighbours; appending each of these resultant children nodes to the **open** list.
4. Next, we loop through each of the newly created child nodes.
    a. If the node has previously been visited, we disregard it;
    b. Else, we append it to the **open** list
5. Once all child nodes have been checked, we remove the **current_node** from the **open** list and add it to the **closed** list.
6. Finally, we return to step 1 and repeat this loop until we find the solution.

Pseudocode for the Depth First Search algorithm follows almost exactly, with one minor modification. In step 3, instead of **appending** the **child** nodes to the **open** list, we must **insert** them to the front of the **open** list; this will make the search algorithm take the DFS path outlined in the Figure above.

A Hint for the Assignments A* Search Implementation

In order to implement the A* search algorithm required in your Towers of Hanoi assignment you must define an appropriate heuristic function. One such heuristic function is as follows:

To calculate the heuristic:

1. Add the number of rings/discs **not** in the right-most stack.
2. Then, add 2 for each ring/disc in the right-most stack, where there is a bigger disc in the other stacks.

The reasoning behind part 1 of this heuristic, is that for each disc not on the right, you will have to do at least one move to get it into the correct stack.

The reasoning behind part 2 of this heuristic, is that for each disc on the right where there is a bigger disc in the other stack, you would first have to move the right-most disc out of the way and then later move it back, hence adding 2.

Utilizing this heuristic function should allow you A* search function to solve the Towers of Hanoi problem optimally.

Lab Instructions

Create a new file named **lab_5** in your working directory.

Launch Jupyter notebook from within Anaconda or open your preferred python IDE.

Ensure that you follow the file structure outline below to correctly structure your code. The final code structure of this file is:

- class Node
  - __init__
  - create_child
  - move_ring
  - expand_node
  - is_correct
  - print_stacks
- class Search
  - depth_first_search
  - path_trace
- Main code

Node Class

Define the **Node** class, this will store the current nodes, stack information, as well as all children nodes.

```
class Node:
    def __init__(self, stacks=None, ring_count=3, stack_count=3, initial_stack_index=0):
        # List to store child nodes
        self.children = []
        # Variable, to store parent node (note: the root nodes parent is "None")
        self.parent = None
        # Current state for this nodes Stacks/Poles/Columns
        self.stacks = stacks
        # Storing the total number of rings for later use
        self.ring_count = ring_count

        # We use this to initialize the root stacks
        # This is only executed for the root node, as self.stacks is None
        if self.stacks is None:
            # Create a List of Empty Lists of length stack_count
            # E.g. if stack_count = 3
            #      self.stacks will be [[],[],[]]
            self.stacks = [[] for i in range(stack_count)]
            for i in range(self.ring_count):
                # This will loop over the total number of specified rings, adding each
                # ring to the top of the stack at the index initial_stack_index
                self.stacks[initial_stack_index].append(i)
```

Define the **create_child** function; this takes a Nodes' stack after a move has been completed in **move_ring** as input and then creates the new Node and sets the parent/children parameters.

```
    def create_child(self, stacks):
        # Create a child Node object using the input stacks
        child = Node(stacks, ring_count=self.ring_count)
        # Store the current node as the parent of the child node
        child.parent = self
        # Store the child node in the children list of the current node
        self.children.append(child)
```

Define the **move_ring** function; this will check a move against our problem rule set. If the move is valid it will create the new **child** Node.

```
    def move_ring(self, from_stack, to_stack):
        # We need to check,
```

```
        #   - if the stack we are moving FROM is empty:
        #       - "len(self.stacks[from_stack])" will be False if the stack is empty
        #
        #   - if the stack we are TO from is empty:
        #       - This is for if we select a initial_stack_index != 0
        #       - If we don't check this, it will throw an error if we try to "pop" off an empty list
        #
        #   - and if the disc we are moving (self.stacks[from_stack][0]) is
        #     smaller than the disc at the move location (self.stacks[to_stack][0])
        if len(self.stacks[from_stack]) and (not self.stacks[to_stack] or self.stacks[to_stack][0]
> self.stacks[from_stack][0]):
            # We use deepcopy to create a stack_copy we can modify without changing the
original
            stacks_copy = deepcopy(self.stacks)
            # We place the disc on the TOP of the specified stack, also removing it from the
original stack with "pop"
            # Note: "Insert" puts at FRONT of list
            #       and "Append" puts at END of list
            stacks_copy[to_stack].insert(0, stacks_copy[from_stack].pop(0))
            # Create the child, using the newly moved stack
            self.create_child(stacks_copy)
```

Define the **expand_node** function; as previously discussed, this will check all possible move variations; utilizing the checks in **move_ring** to verify validity

```
    def expand_node(self):
        # To expand out node, we must check all possible movements
        number_of_stacks = len(self.stacks)

        for i in range(number_of_stacks):
            for j in range(number_of_stacks):
                # This will attempt to move the top disc of all stacks to all other stacks
                # Our move function will decide whether this move is possible or not
                self.move_ring(i, j)
```

Define **is_correct**; this function will loop from 0 to 5 checking the ring order in the last **stack** from our **self.stacks** list; if this succeeds, we know we have achieved the goal state.

```
    def is_correct(self):
        # In order for our node to be correct,
        #   we know that all discs must be arranged in the final stack,
        #   and they must be in the correct order 0->5
```

```
        for i in range(self.ring_count):
            # Loop from 0->ring_count
            try:
                # If the disc in the ith position of the last stack = i from our loop
                if self.stacks[len(self.stacks)-1][i] == i:
                    # we do nothing and continue checking
                    pass
            except:
                # An exception will be triggered if self.stacks[len(self.stacks)-1][i] doesn't exist
                # This means that the there is no disc in the ith position of the last stack.
                # In this case we know that we havn't found the objective and can return False.
                # i.e. if i=5 and the last_stack = [0,1,2,3,4]
                #      when we check last_stack[i] it will enter this exception as that index doesn't
exist.
                return False
        # If we reach this point, we know that we have found the objective!
        return True
```

define **print_stacks**, this is a utility function which will print the Towers of Hanoi stacks in a formatted easy to read way. The **delay_increment** can be modified to increase/decrease the speed at which nodes are traversed.

```
    def print_stacks(self, delay_increment=0.03):
        # These two imports are based on your environment
        from os import system  # This is for terminal/command prompt
        from IPython.display import clear_output   # This is for Jupyter Notebooks

        # This function is just a utility to display the stacks in a easy to read way
        max_height = self.ring_count

        for ring_height in range(max_height, 0, -1):
            for stack_index in range(len(self.stacks)):
                if len(self.stacks[stack_index]) >= ring_height:
                    print(self.stacks[stack_index]
                          [-ring_height], " ", end="")
                else:
                    print("   ", end="")
            print("")
        sleep(delay_increment)

        # Select one of these based on your environment
        # Leave these commented out if you want to view all node configurations
```

```
    # system('clear')  # This will clear a terminal/command prompt output
    # clear_output(wait=True)  # This will clear a Jupyter Notebooks output
```

Define the **Search** class; as in our previous lab tasks, this class will hold all of the search algorithms. Within this class define **depth_first_search**; this is the code to execute our DFS pseudocode. It is almost identical to the **breadth_first_search** function we previously implemented, with the slight modification to how child nodes are added to **open_list** as discussed earlier.

```
class Search:
    def depth_first_search(self, root):
        # List to contain open nodes
        open_list = []
        # Set to contain visited nodes
        visited = set()
        # Add root node as open
        open_list.append(root)
        # Add root node as a visited state
        visited.add(tuple(map(tuple, root.stacks)))

        while(True):
            # Get next node to search from the top of the list of open nodes
            current_node = open_list.pop(0)

            # Check if the current node is the goal state
            if current_node.is_correct():
                # If we have found the goal state, store the path to the current state
                path_to_solution = self.path_trace(
                    current_node)
                return path_to_solution, len(visited)

            # If current node is not the goal state, then find its neighbouring nodes
            current_node.expand_node()

            # Loop through all nodes neighbouring the current node
            for current_child in current_node.children:

                # If neighbouring child hasn't previously been visited
                if (not tuple(map(tuple, current_child.stacks)) in visited):
                    # Add neighbouring child to list of open nodes
                    # Using the "Insert" Function puts the current child to the front of the open_list
                    # This will make it Depth First Search
                    open_list.insert(0, current_child)
```

```
                # Add current child to set of visited nodes
                visited.add(tuple(map(tuple, current_child.stacks)))
```

Define **path_trace**; this function is identical to that of our previous lab tasks. It simply iterates through the Nodes' **parents** to trace the goal path back to the root node.

```
    def path_trace(self, node):
        # Store the input node
        current = node
        # Create a list named path, this will store all nodes in the path
        path = []
        # Append the initial node to the path list
        path.append(current)
        # Loop while our current node isn't the root node (as our root node's parent is "None")
        while current.parent != None:
            # Set current node to the parent of the previous node
            current = current.parent
            # Append the current node to the path list
            path.append(current)
        # Return the final path from root node to goal node
        return path
```

Finally, we enter the main block of code where our DFS implementation will be called.

```
if __name__ == "__main__":
    # Initialize our puzzle, we specify 6 rings, 3 stacks
    # and that we want the rings to be generated on stack 0
    root = Node(ring_count=6, stack_count=3, initial_stack_index=0)
    search = Search()

    # Capture the search start time
    time_start = time()
    # Execute the search and store the returned variables
    path_to_solution, visited_nodes_count = search.depth_first_search(root)
    # Capture the search end time
    time_end = time()

    # Reverse our stored path so that we can view it in correct order
    path_to_solution.reverse()

    # Display the stacks at each node in our solution
```

```
for node in path_to_solution:
    # Modify the delay_increment in this function call to increase/decrease
    # the speed at which each node is displayed
    node.print_stacks(delay_increment=0.01)

# Print out our results
print("Total Nodes Visited During Search:", visited_nodes_count)
print("Final Path Node Count            :", len(path_to_solution)-1)
print("Total Elapsed Search Time        : {:.5f} s".format(
    time_end-time_start))
```

You may now run this code; upon doing so you should see the path from the root to the objective node be printed out.

(Note: un-commenting the appropriate “clear” line in the **print_stacks** function will alter how the puzzle is displayed.)

Utilizing the information in this Lab, you can now implement your own Towers of Hanoi solver to include both Breadth First Search and A* Search as required in the Assignment.