SSN College of Engineering

Department of Computer Science and Engineering

CS1504 — Artificial Intelligence

2021 – 2022

Assignment — 07
(Expression Search)

September 11, 2021

Problem Statement

The objective is to use the given six numbers to arithmetically calculate a randomly chosen number. Only the four arithmetic operations of addition, subtraction, multiplication and division may be used, and no fraction may be introduced into the calculation. Each number may be used at most once. The initial six numbers should be taken as input from the user. The target number should be generated randomly from 100 to 999. If the exact expression is not found, the program should print out the expression that evaluates to a value that is the closest to the target value

State Space Formulation

State: An expression comprising between 0 and 6 of the given operands (with operators in between)

State Representation: Expressions are represented as expression trees

Initial State: 0 operands and 0 operators

Actions: Choose an operand, operator and order of operation

Goal State: Expression evaluates to target value and all operands are used

Search Strategy: Breadth First Search

Python Program Code

1. StateFormulation.py - Script to formulate the state space and instantiate a problem case

```
from ExpressionTree import ExpressionTree, ExpressionNode
```

```
def get_init_state(operand_space):
    forest = []
    for operand in operand_space:
        node = ExpressionNode(operand, False)
        tree = ExpressionTree(
            operand_space=operand_space,
            root_node=node
        )
        forest.append(tree)
    return forest



# Find the next states
def get_next_states(state, operand_space):
    # State must be a single expression tree
    mergeables = get_init_state(operand_space)
    next_states = []
    for other_state in mergeables:
        for operation in ExpressionNode.operations:
            if operation in ExpressionNode.operations_commutative:
                # Commutative operation
                if ExpressionTree.is_merge_valid(operation, state,
other_state):
                    operation_node = ExpressionNode(operation, True)

next_states.append(ExpressionTree.merge_trees(operation_node, state,
other_state))
            else:
                # Not a commutative operation
                # Order 1
                if ExpressionTree.is_merge_valid(operation, state,
other_state):
```

```
                    operation_node = ExpressionNode(operation, True)

next_states.append(ExpressionTree.merge_trees(operation_node, state,
other_state))
                # Order 2
                if ExpressionTree.is_merge_valid(operation,
other_state, state):
                    operation_node = ExpressionNode(operation, True)

next_states.append(ExpressionTree.merge_trees(operation_node,
other_state, state))
    return next_states



def is_operand_space_exhausted(state):
    return all(state.bitmask)

def is_goal_reached(state, goal_value):
    return is_operand_space_exhausted(state) and
state.evaluation==goal_value



"""
# Testing state generation
sample_state = [4, 5, 6, 3, 4, 5, 6, 5]
sample_state = [1, 5, 7, 2, 3, 3, 6, 4]
print(get_next_states(sample_state))
"""
```

2. ExpressionTree.py - Script to perform represent expression as tree

```
from copy import deepcopy
```

```
class ExpressionNode:

    operations = [ '+', '-', '*', '/' ]
    operations_commutative = ['+', '*']

    def __init__(self, value, assert_operation=True):
        self.is_operand = value not in self.operations
        if assert_operation:
            assert self.is_operand == False
        # Set node values
        self.value = value
        self.left_child = None
        self.right_child = None

    def get_bitmask(self, value, operand_space):
        return [ operand==x for x in operand_space ]



class ExpressionTree:

    """
    Nodes represent sequence of numbers (tree-structure)
    Edges represent an operation (+, -, *, /)
    """

    @classmethod
    def merge_bitmasks(cls, bitmask_1, bitmask_2):
        return [ x or y for x,y in zip(bitmask_1, bitmask_2) ]

    @classmethod
```

```
    def evaluate(cls, operation, lhs, rhs):
        if operation == '+':
            return lhs + rhs
        elif operation == '-':
            return lhs - rhs
        elif operation == '*':
            return lhs * rhs
        elif operation == '/':
            return lhs // rhs

    @classmethod
    def is_merge_valid(cls, operation, lhs, rhs):
        if operation == '/' and lhs.evaluation % rhs.evaluation != 0:
            # print("F1")
            return False
        if lhs.operand_space != rhs.operand_space:
            # print("F2")
            return False
        # NAND-operation
        # Same operand must NOT have been used in both already
        for x_bit, y_bit in zip(lhs.bitmask, rhs.bitmask):
            if x_bit and y_bit:
                # print("F3")
                return False
        return True

    @classmethod
    def merge_trees(cls, operation_node, lhs, rhs):
        """
        lhs, rhs are trees
        """
```

```
        # Ensure the nodes can be merged
        assert cls.is_merge_valid(operation_node.value, lhs, rhs) ==
True
        # Merge nodes
        operation_node.left_child = lhs
        operation_node.right_child = rhs
        # Make new tree
        lhs = deepcopy(lhs)
        rhs = deepcopy(rhs)
        new_tree = ExpressionTree(
            operand_space=lhs.operand_space,
            root_node=operation_node,
            bitmask=cls.merge_bitmasks(lhs.bitmask, rhs.bitmask),
            evaluation=cls.evaluate(operation_node.value,
lhs.evaluation, rhs.evaluation)
        )
        return new_tree


    def __init__(self, operand_space, root_node, bitmask=None,
evaluation=None):
        """
        For constructing trees from scratch, root_node is assumed to
be a single node
        Bitmask is computed. Expression result is computed
        For constructing trees from two other trees, bitmask must be
supplied through the merge
        operation. Expression evaluation must be passed
        """
        self.operand_space = operand_space
        self.root = root_node
        # Bitmask set
        if bitmask is None:
```

```
            self.bitmask = self.get_bitmask(self.root)
        else:
            self.bitmask = bitmask
        # Evaluation result
        if evaluation is None:
            self.evaluation = self.root.value
        else:
            self.evaluation = evaluation

    def get_bitmask(self, operand_node):
        return [ operand_node.value==x for x in self.operand_space ]

    def display(self):
        # Left recursion
        print("(", end="")
        if self.root.left_child is None:
            pass
        else:
            self.root.left_child.display()
        # Current value
        print(self.root.value, end="")
        # Right recursion
        if self.root.right_child is None:
            pass
        else:
            self.root.right_child.display()
        print(")", end="")

"""
def run_test():
    operands = [1, 2, 3, 5]
```

```
    root_node = ExpressionNode(5, False)
    tree_1 = ExpressionTree(
        operands,
        root_node
    )
    root_node = ExpressionNode(1, False)
    tree_2 = ExpressionTree(
        operands,
        root_node
    )
    operation = ExpressionNode('+', True)
    tree = ExpressionTree.merge_trees(operation, tree_1, tree_2)
    tree.display()
    print(operands)
    print(tree.bitmask)
    print(tree.evaluation)

run_test()
"""
```

3. Queue.py - Queue data structure for BFS

```
class Queue:
    # [HEAD, ....... , TAIL]
    def __init__(self, data_list=None):
        self.data = list()
        self.size = 0
        if data_list is not None:
            self.data.extend(data_list)
            self.size = len(data_list)
```

```
    def enqueue(self, data_list):
        self.data.extend(data_list)
        self.size += len(data_list)

    def dequeue(self):
        if self.size == 0:
            return None
        self.size -= 1
        return self.data.pop(0)

    def is_empty(self):
        return self.size==0
```

4. BreadthFirstSearch.py - Implementation of BFS strategy

```
from Queue import Queue
from numpy import inf
from copy import deepcopy

from StateFormulation import *


def search(operand_space, goal_value):

    # Queue of fringe states and their respective paths
    state_queue = Queue(get_init_state(operand_space))
    # Node Counters
    visited_cnt = 0
    generated_cnt = 0
    closest_result = ExpressionTree(
        operand_space=operand_space,
```

```
        root_node=ExpressionNode(inf, False),
        bitmask=[],
        evaluation=inf
    )
    while(not state_queue.is_empty()):
        # PRE-VISIT
        state = state_queue.dequeue()
        # VISIT
        if is_operand_space_exhausted(state):
            best_diff = abs(goal_value-closest_result.evaluation)
            curr_diff = abs(goal_value-state.evaluation)
            if curr_diff == 0:
                # Goal hit
                return state
            elif curr_diff < best_diff:
                # New nearer state
                closest_result = deepcopy(state)
        # POST-VISIT
        # Find fringe and add to queue
        fringe = get_next_states(state, operand_space)
        state_queue.enqueue(fringe)

    if closest_result.evaluation == inf:
        return False
    else:
        return closest_result

"""
search(
    operand_space=[4, 8, 9],
    goal_value=18
```

```
).display()
"""
```

5. main.py - Driver for running the search

```
import BreadthFirstSearch as BFS


operand_space = list(map(int, input("Enter Operands as
space-separated Integers\n").split()))
target_val = int(input("Enter Target Expression Value: "))


result_tree = BFS.search(operand_space, target_val)
# Display result
if result_tree.evaluation==target_val:
   print("Exact expression found")
else:
   print("Nearest expression found. Expression result:",
result_tree.evaluation)
result_tree.display()
print()
```

Sample Output (both cases depicted)

```
Enter Operands as space-separated Integers
8 4 9
Enter Target Expression Value: 18
Exact expression found
(((8)/(4))*(9))
(base) karthikd@Karthik-DEBIAN:~/Workspace/Computer
Enter Operands as space-separated Integers
8 4 9
Enter Target Expression Value: 12
Nearest expression found. Expression result: 13
(((8)-(4))+(9))
```