SSN College of Engineering

Department of Computer Science and Engineering

CS1504 — Artificial Intelligence

2021 – 2022

Assignment — 04
(State Space Search — Decantation Problem)

August 16, 2021

Problem Statement

You are given an 8-litre jar full of water and two empty jars of 5- and 3-litre capacity. You have to get exactly 4 litres of water in one of the jars. You can completely empty a jar into another jar with space or completely fill up a jar from another jar.

1. Formulate the problem: Identify states, actions, initial state, goal state(s). Represent the state by a 3-tuple. For example, the initial state state is (8,0,0). (4,1,3) is a goal state (there may be other goal states also).
2. Use a suitable data structure to keep track of the parent of every state. Write a function to print the sequence of states and actions from the initial state to the goal state.
3. Write a function next states(s) that returns a list of successor states of a given state s.
4. Implement iterative deepening algorithm and bidirectional search algorithm to search the state space graph for a goal state that produces the required sequence of pouring's

Responses

1. Formulated below (State Space Formulation)
2. A stack data structure is used to perform Depth-limited search algorithm as part of the Iterative Deepening Search (Stack.py)
   A queue data structure is used to perform Breadth-First-Search as part of the Bidirectional Search (Queue.py)
   A dictionary with first explored parent of each state is maintained
3. Function implemented as *get_next_state(state)* (StateFormulation.py)

(*Link to **repl.it** implementation repository*)

State Space Formulation

3-tuple Representation

(a, b, c) where,

a is the amount of water in the 8-liter jar

b is the amount of water in the 5-liter jar

c is the amount of water in the 3-liter jar

Initial State : (8, 0, 0)

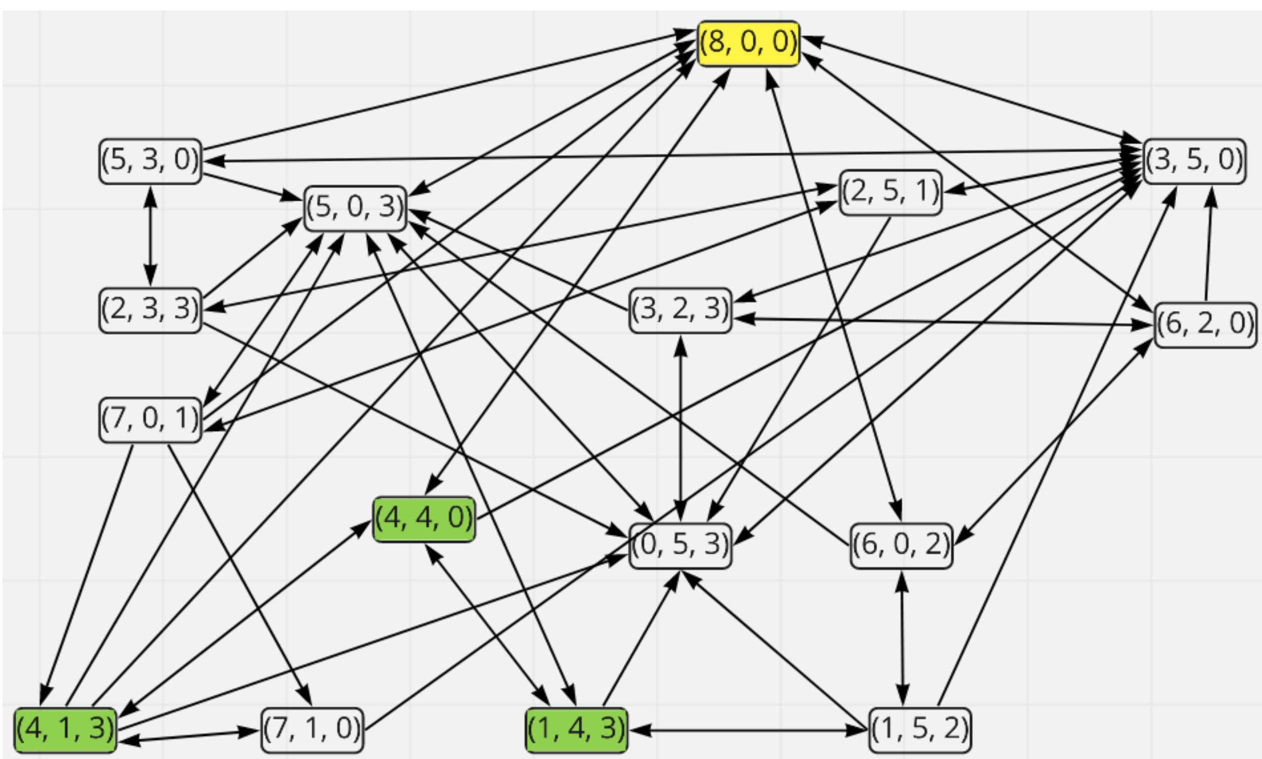
Allowed actions to progress to Next State :

1. Empty a jar into another with empty space
2. Fill up a jar completely from another jar

Constraint: Sum of quantity of water over all jars is always equal to 8 liters

Goal State : One of the three jars contains exactly 4L of water

State Transition Diagram



Python Program Code

1. Stack.py - Script for Stack ADT

```
class Stack:
   # [TOP, ....... , BOTTOM]
   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 push(self, data_list):
       self.data = data_list + self.data
       self.size += len(data_list)

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

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

2. Queue.py - Script for Queue ADT

```
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 get_contents(self):
        return self.data

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

3. StateFormulations.py - Functions and constants to generate and evaluate states

```
# CONSTANTS
CAPACITY = (8,5,3)
NUM_JUGS = 3
INITIAL_STATE = (8,0,0)
GOAL_STATES = [(4,1,3), (4,4,0), (1,4,3)]

def get_next_states(state):

    # Transfer water from jug 'from_' to 'to'
    def transfer(from_, to):
        # Find bottleneck
        space = CAPACITY[to] - state[to]
        water = state[from_]
        transit = min(space, water)
```

```python
        # Perform transfer
        new_state = list(state)
        new_state[to] += transit
        new_state[from_] -= transit
        return tuple(new_state)

    # From i to j
    # Check all possible combinations
    result = list()
    for i in range(NUM_JUGS):
        for j in range(NUM_JUGS):
            if (i==j):
                # Self-transfer is meaningless
                continue
            if CAPACITY[j]==state[j]:
                # Cannot fill more
                # Destination full
                continue
            if state[i]==0:
                # Nothing to transfer
                # Source empty
                continue
            result.append(transfer(i,j))

    return result

def is_goal_state(state):
    return 4 in state

# Compare the current state of `this` direction search
# to the fringe states of `that` direction search
def intersection_test(this_state, that_fringe):

    def are_states_same(state_A, state_B):
        for i in range(NUM_JUGS):
            if(state_A[i]!=state_B[i]):
                return False
        return True
```

```
    for state in that_fringe:
        if are_states_same(state, this_state):
            return True
    return False
```

4. IterativeDeepening.py - Script to perform Iterative Deepening search

```

from Stack import *
from StateFormulation import *

def deduce_path(state, parents):

    def deduce_path_rec(state, path_seq):
        this_parent = parents[state]
        # Reached the root node
        if this_parent is None:
            return path_seq
        # Prepend node
        path_seq = [this_parent] + path_seq[:]
        return deduce_path_rec(this_parent, path_seq)

    return deduce_path_rec(state, [])


def make_path_string(path, depths):
    string_path = list()
    for state in path:
        string_path.append("{state} (Depth:
{depth})".format(state=state, depth=depths[state]))
    return string_path


def search(num_solns_reqd):

    def search_depth():

        state_space = Stack([INITIAL_STATE])
        depth_track = Stack([0])
        goal_states = list()
```

```
       explored_states = set()
       parents = {INITIAL_STATE: None}
       depths = {INITIAL_STATE: 0}

       while not state_space.is_empty():
           # Get next state and its path
           state = state_space.pop()
           curr_depth = depth_track.pop()
           # Skip iteration if state already explored
           if state in explored_states:
               continue
           if curr_depth > limit:
               continue
           explored_states.add(state)
           # Check if the state is a goal-state
           if is_goal_state(state):
               goal_states.append(state)
           # Find the next states
           fringe = get_next_states(state)
           state_space.push(fringe)
           # Add parent and depth of each fringe state, if not
already added
           fringe_depths_list = list()
           for new_state in fringe:
               if new_state not in parents:
                   parents[new_state] = state
                   depths[new_state] = curr_depth+1
                   fringe_depths_list.append(curr_depth+1)
               else:
                   fringe_depths_list.append(depths[new_state])
           depth_track.push(fringe_depths_list)

       return goal_states, explored_states, parents, depths

   limit = -1
   num_solns = 0
   while(num_solns<num_solns_reqd):
       limit += 1
       goal_states, explored_states, parents, depths =
search_depth()
```

```
        num_solns = len(goal_states)
        print("{num_solns} solutions found with depth limit
{limit}".format(num_solns=num_solns, limit=limit))
    return goal_states, explored_states, parents, depths
```

5. BidirectionalBFS.py - Script to perform Bidirectional BFS

```
from Queue import Queue
from StateFormulation import *

def deduce_path(connecting_state, f_parents, r_parents):

    def deduce_path_rec(f_state, r_state, path_seq, f_depth,
r_depth):
        f_parent = f_parents[f_state] if f_state is not None else
None
        r_parent = r_parents[r_state] if r_state is not None else
None
        recurse = False

        if f_parent is not None:
            path_seq = [f_parent] + path_seq
            f_depth += 1
            recurse = True
        if r_parent is not None:
            path_seq = path_seq + [r_parent]
            r_depth += 1
            recurse = True

        if recurse:
            return deduce_path_rec(f_parent, r_parent, path_seq,
f_depth, r_depth)
        else:
            return path_seq, f_depth, r_depth

    return deduce_path_rec(connecting_state, connecting_state,
[connecting_state], 0, 0)



def search(goal_states):
```

```
def search_chosen_goal(goal_state):

    f_state_space = Queue([INITIAL_STATE])
    f_explored_states = set()
    f_parents = {INITIAL_STATE: None}

    r_state_space = Queue([goal_state])
    r_explored_states = set()
    r_parents = {goal_state: None}

    while(not f_state_space.is_empty() or not
r_state_space.is_empty()):
        # Forward Search
        f_state = f_state_space.dequeue()
        if f_state not in f_explored_states:
            # Explore now
            f_explored_states.add(f_state)
            if intersection_test(f_state,
r_state_space.get_contents()):
                return f_explored_states, r_explored_states,
f_state, f_parents, r_parents
            fringe = get_next_states(f_state)
            f_state_space.enqueue(fringe)
            for new_state in fringe:
                if new_state not in f_parents:
                    f_parents[new_state] = f_state

        # Reverse Search
        r_state = r_state_space.dequeue()
        if r_state not in r_explored_states:
            # Explore now
            r_explored_states.add(r_state)
            if intersection_test(r_state,
f_state_space.get_contents()):
                return f_explored_states, r_explored_states,
r_state, f_parents, r_parents
            fringe = get_next_states(r_state)
            r_state_space.enqueue(fringe)
            for new_state in fringe:
                if new_state not in r_parents:
```

```
                        r_parents[new_state] = r_state

        return False

    results = dict()
    for goal in goal_states:
        result = search_chosen_goal(goal)
        if not result:
            continue
        results[goal] = result
    return results
```

6. main.py - Driver program to implement and summarize the search techniques

```
import IterativeDeepening
import BidirectionalBFS
from StateFormulation import *


line = "----------------------------"


if __name__ == '__main__':

    num_expected_solns = 3
    print("\n"+line)
    print("ITERATIVE DEEPENING Search\n")
    goal_states, explored_states, parents, depths =
IterativeDeepening.search(num_expected_solns)

    print("\nDISTINCT EXPLORED STATES COUNT: ", len(explored_states))

    print("\nGOAL STATES COUNT: ", len(goal_states))

    print("\nINITIAL STATE")
```

```
    print(INITIAL_STATE)

    print("\nGOAL STATES")
    for state in goal_states:
        print(state)

    print("\nEXPLORED STATES")
    for state in explored_states:
        print(state)

    for state in goal_states:
        print("\nPATH to reach", state)

print("\n".join(IterativeDeepening.make_path_string(IterativeDeepeni
ng.deduce_path(state, parents), depths)))
        print(state, '--> GOAL STATE')

    # Bidirectional Search
    print("\n"+line)
    print("BIDIRECTIONAL Search\n")

    print("INITIAL STATE")
    print(INITIAL_STATE)
    results = BidirectionalBFS.search(GOAL_STATES)
    for goal_state in results:
        # Unpack result parameters
        f_explored_states, r_explored_states, conn_state, f_parents,
r_parents = results[goal_state]
        print("\n"+line)
        print("For GOAL STATE", goal_state)

        explored_states = f_explored_states.union(r_explored_states)
        print("\nDISTINCT EXPLORED STATES COUNT: ",
len(explored_states))
        print("\nEXPLORED STATES")
        for state in explored_states:
            print(state)

        print("\nPATH TAKEN")
```

```
        goal_path, f_depth, r_depth =
BidirectionalBFS.deduce_path(conn_state, f_parents, r_parents)
        for state in goal_path:
            if(state==conn_state):
                print(state, '--> CONNECTING STATE')
            elif(state==goal_state):
                print(state, '--> GOAL STATE')
            else:
                print(state)
```

(Output on next page)

Sample Output

```
----------------------------
ITERATIVE DEEPENING Search

0 solutions found with depth limit 0
0 solutions found with depth limit 1
0 solutions found with depth limit 2
0 solutions found with depth limit 3
0 solutions found with depth limit 4
0 solutions found with depth limit 5
1 solutions found with depth limit 6
2 solutions found with depth limit 7
2 solutions found with depth limit 8
3 solutions found with depth limit 9

DISTINCT EXPLORED STATES COUNT:  15

GOAL STATES COUNT:  3

INITIAL STATE
(8, 0, 0)

GOAL STATES
(4, 1, 3)
(4, 4, 0)
(1, 4, 3)

EXPLORED STATES
(6, 2, 0)
(2, 3, 3)
(3, 2, 3)
(0, 5, 3)
(3, 5, 0)
(7, 0, 1)
(2, 5, 1)
(7, 1, 0)
(1, 4, 3)
(6, 0, 2)
(4, 1, 3)
(5, 0, 3)
(4, 4, 0)
(5, 3, 0)
(8, 0, 0)
```

```
PATH to reach (4, 1, 3)
(8, 0, 0) (Depth: 0)
(5, 0, 3) (Depth: 1)
(5, 3, 0) (Depth: 2)
(2, 3, 3) (Depth: 3)
(2, 5, 1) (Depth: 4)
(7, 0, 1) (Depth: 5)
(7, 1, 0) (Depth: 6)
(4, 1, 3) --> GOAL STATE

PATH to reach (4, 4, 0)
(8, 0, 0) (Depth: 0)
(5, 0, 3) (Depth: 1)
(5, 3, 0) (Depth: 2)
(2, 3, 3) (Depth: 3)
(2, 5, 1) (Depth: 4)
(7, 0, 1) (Depth: 5)
(7, 1, 0) (Depth: 6)
(4, 1, 3) (Depth: 7)
(4, 4, 0) --> GOAL STATE

PATH to reach (1, 4, 3)
(8, 0, 0) (Depth: 0)
(5, 0, 3) (Depth: 1)
(5, 3, 0) (Depth: 2)
(2, 3, 3) (Depth: 3)
(2, 5, 1) (Depth: 4)
(7, 0, 1) (Depth: 5)
(7, 1, 0) (Depth: 6)
(4, 1, 3) (Depth: 7)
(4, 4, 0) (Depth: 8)
(1, 4, 3) --> GOAL STATE
```

```
----------------------------
BIDIRECTIONAL Search

INITIAL STATE
(8, 0, 0)

----------------------------
For GOAL STATE (4, 1, 3)

DISTINCT EXPLORED STATES COUNT:  4

EXPLORED STATES
(0, 5, 3)
(4, 1, 3)
(8, 0, 0)
(3, 5, 0)

PATH TAKEN
(8, 0, 0)
(3, 5, 0)
(0, 5, 3) --> CONNECTING STATE
(4, 1, 3) --> GOAL STATE

----------------------------
For GOAL STATE (4, 4, 0)

DISTINCT EXPLORED STATES COUNT:  3

EXPLORED STATES
(4, 4, 0)
(8, 0, 0)
(3, 5, 0)

PATH TAKEN
(8, 0, 0)
(3, 5, 0) --> CONNECTING STATE
(4, 4, 0) --> GOAL STATE
```

```
----------------------------
For GOAL STATE (1, 4, 3)

DISTINCT EXPLORED STATES COUNT:  4

EXPLORED STATES
(1, 4, 3)
(0, 5, 3)
(8, 0, 0)
(3, 5, 0)

PATH TAKEN
(8, 0, 0)
(3, 5, 0)
(0, 5, 3) --> CONNECTING STATE
(1, 4, 3) --> GOAL STATE
```