**First problem:**

The Water Jug Problem: Given two jugs without any measuring markings on it, a 4-litre one and a 3-litre one and an infinite source of water, how can you get exactly 2 litre of water in the 4-litre jug? Based on the given Python framework, create a model representation for this problem and apply proper informed/uninformed search strategy.

**Code:**

from search import Problem

class JugProblem(Problem):

def result(self, state, action):

return action

def value(self, state):

pass

def \_\_init\_\_(self, initial, goal):

self.goal = goal

self.initial = initial

self.visited\_states = []

Problem.\_\_init\_\_(self, self.initial, self.goal)

def \_\_repr\_\_(self):

return "< State (%s, %s) >" % (self.initial, self.goal)

def goal(self, state):

if (state[0]==2 or self.goal[0]==3):

return True

else:

return False

def actions(self, cur\_state):

actions = []

self.visited\_states.append(cur\_state)

#Pour water in the first Jug

new\_state = (4, cur\_state[1])

actions.append(new\_state)

#Pour water in the second Jug

new\_state = (cur\_state[0], 3)

actions.append(new\_state)

#Pour water from the first jug in the second jug

if cur\_state[0] + cur\_state[1] <= 3:

new\_state = (0, cur\_state[0] + cur\_state[1])

else:

new\_state = (0, 3)

if cur\_state[0] - (3 - cur\_state[1]) >= 0:

new\_state = (cur\_state[0] - (3 -cur\_state[1]), new\_state[1])

else:

new\_state = (0,new\_state[1])

if cur\_state[1] != 3:

actions.append(new\_state)

#Pour water from the second jug in the first jug

if cur\_state[0] + cur\_state[1] <= 4:

new\_state = (cur\_state[0] + cur\_state[1], 0)

else:

new\_state = (4, 0)

if cur\_state[1] - (4 - cur\_state[0]) >= 0:

new\_state = (new\_state[0], cur\_state[1] - (4 -cur\_state[0]))

else:

new\_state = (new\_state[0], 0)

if cur\_state[0] != 4:

actions.append(new\_state)

#Empty the first jug

new\_state = (0, cur\_state[1])

if cur\_state[0] != 0:

actions.append(new\_state)

#Empty the second jug

new\_state = (cur\_state[0], 0)

if cur\_state[1] != 0:

actions.append(new\_state)

return actions

A state is composed of two elements, first represents the water in the first jug and the other represents the water in the second jug. The initial state is (0, 0) because both the jugs are empty, and the final state is where the first jug (capacity 4L) contains 2L of water.

Pour water in the first Jug: we fill the 4L jug with water(the maximum of water that can be poured here). It has no effect on the second jug.

Pour water in the second Jug: we fill the 3L jug with water(the maximum of water that can be poured here). It has no effect on the first jug.

Pour water from the first jug in the second jug: We fill the second jug or the first jug is emptied.

Pour water from the second jug in the first jug: We fill the first jug or the second jug is emptied.

Empty the first jug: Empties the first jug.

Empty the second jug: Empties the second jug.

The goal function is checking if our goal is reached, we must obtain 2L of water in the first jug(the 4L capacity jug) and we don’t care about the second one, as example it can be filled(3L).

**Second problem:**

Based on the 8-puzzle problem from the provided frame-work build a model representation of the 15-puzzle problem and create two heuristic functions (different from those presented at the laboratory) to solve the problem by applying A\* search strategy.

**Code:**

import math

from search import Problem

class FifteenPuzzle(Problem):

def \_\_init\_\_(self, initial, goal):

self.goal = goal

self.initial = initial

Problem.\_\_init\_\_(self, initial, goal)

def blk\_square(self, state):

return state.index(0)

def actions(self, state):

possible\_actions = ['LEFT', 'RIGHT', 'UP', 'DOWN']

index\_blk\_square = self.blk\_square(state)

if index\_blk\_square % 4 == 3:

possible\_actions.remove('RIGHT')

if index\_blk\_square % 4 == 0:

possible\_actions.remove('LEFT')

if index\_blk\_square < 4:

possible\_actions.remove('UP')

if index\_blk\_square > 11:

possible\_actions.remove('DOWN')

return possible\_actions

def result(self, state, action):

index\_blk\_square = self.blk\_square(state)

new\_state = list(state)

delta = { 'LEFT':-1, 'RIGHT':1, 'UP':-4, 'DOWN':4}

neighbor = index\_blk\_square + delta[action]

new\_state[index\_blk\_square], new\_state[neighbor] = new\_state[neighbor], new\_state[index\_blk\_square]

return tuple(new\_state)

def goal(self, state):

return state == self.goal

def is\_solvable(self, state):

inv = 0

for i in range(len(state)):

for j in range(i, len(state)):

if state[i] > state[j] != 0:

inv += 1

return inv % 2 == 0

# Euclidian Method

def h(self, node):

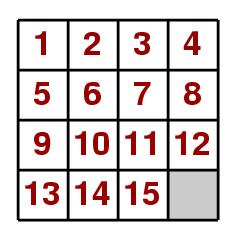
return sum(math.sqrt(pow(s // 4 - g // 4,2) + pow(s % 4 - g % 4,2)) for (s,g) in zip(node.state, self.goal))

# Manhattan Method

#def h(self, node):

#return sum( abs( s // 4 - g // 4) + abs( s % 4 - g % 4) for (s,g) in zip(node.state, self.goal) )

We have 16 numbers asigned for tiles. The empty tile is 0. The goal state is (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 0).



“UP, DOWN, LEFT, RIGHT” are all the possible actions. Depends on the case of the blank tile which moves are impossible, for example if the index of the tile is 15 it is impossible to move it down or right.

The goal function verifies if the current state is equal to the goal state.

This function “is\_solvable” checks the given state if it can be solved.