CSCI 3202 Intro to Artificial Intelligence Problem Set 1

Luke Meszar

September 25, 2017

```
import networkx as nx
import itertools

def generate_nodes(size_list):
    can1 = [i for i in range(31,size_list[0]+1)]
    can2 = [i for i in range(31,size_list[1]+1)]
    can3 = [i for i in range(size_list[2]+1)]
    can4 = [i for i in range(size_list[3]+1)]
    nodes = list(itertools.product(can1,can2,can3,can4)) #take cartesian product
    correct_nodes = []
    for node in nodes:
        if sum(list(node)) == 80: #make sure the node only has 80 quarts of
        #milk meaning it is valid
            correct_nodes.append(node)
    milk_cans_graph.add_nodes_from(correct_nodes)
    generate_edges(correct_nodes,size_list)

def generate_edges(nodes,size_list):
    for i in range(len(nodes)):
        for j in range(len(nodes)):
            if (i != j and are_connected(nodes[i],nodes[j],size_list)):
                milk_cans_graph.add_edge(nodes[i],nodes[j]) #create an edge
                #if we have determined there is a move that transfers between
                #both states

def are_connected(node1,node2,size_list):
    for i in range(4):
        amount_to_pour  = node1[i] #how much we can pour from can i
        for j in range(4):
            if i != j: #don't want to pour a can into itself
                new_node_list = list(node1) #get mutable copy of node1
                actual_amount_poured = amount_to_pour
                if amount_to_pour + new_node_list[j] > size_list[j]: #there is
                #more to pour then there is capacity
                    actual_amount_poured = size_list[j] - new_node_list[j]
                    new_node_list[j] = size_list[j] #max out can
                else:
                    new_node_list[j] += amount_to_pour #otherwise, pour entirety
                    #of can i into can j
                new_node_list[i] -= actual_amount_poured #subtract how much
                #we have poured
                move_node = tuple(new_node_list)
                if move_node == node2: #compare both nodes
                     return True
    return False
```

```
def dfs_paths(start):#standard iterative DFS that keeps track of paths
    stack = [(start, [start])]
    visited = set()
    while stack:
        (vertex, path) = stack.pop()
        if vertex not in visited:
            if vertex[2] == 2 and vertex[3] == 2: #we have found a solution
                return (path,visited)
            visited.add(vertex)
            for neighbor in milk_cans_graph.neighbors(vertex):
                stack.append((neighbor, path + [neighbor]))

def dfs_traversal(start):#standard iterative DFS does not stop when a
#goal state is found this finds the strongly connected component from
#the start node
    stack = [start]
    visited = set()
    while stack:
        vertex = stack.pop()
        if vertex not in visited:
            visited.add(vertex)
            for neighbor in milk_cans_graph.neighbors(vertex):
                stack.append(neighbor)
    print(count_unique(visited)) #size of SCC

def count_unique(visited):
    list_visited = list(visited)

    for i in range(len(list_visited)):
        current_node = list_visited[i]
        if current_node[1] > current_node[0]:
            list_visited[i] = (current_node[1],current_node[0],current_node[2],current_node[3]

    return len(set(list_visited))




def iterative_deepening(start): #iterates through a number of depth limited
#searches until a solution is found
    path = []
    visited = set()
    for depth in range(0,20):
        found = depth_limited_search(start,depth,path)
        if found:
            return found

def depth_limited_search(node,depth,path): #depth limited search
    pathn = path[:] #create copy of path (not reference)
    pathn.append(node)
    if depth == 0 and node[2] == 2 and node[3] == 2:
        print("solution:")
        print(pathn)
```

```
        return node
    if depth > 0:
        for neighbor in milk_cans_graph.neighbors(node):
            found = depth_limited_search(neighbor,depth-1,pathn)
            if found:
                return found



if __name__ == "__main__":
    milk_cans_graph=nx.DiGraph()
    generate_nodes([40,40,5,4])
    (path, visited) = dfs_paths((40,40,0,0))
    print(path)
    print(len(path))

    iterative_deepening((40,40,0,0))
    dfs_traversal((40,40,0,0))
```