Artificial Intelligence Lab

Lab 10: Implementation of block world problem

Tamojit Sarkar
RA1811027010034
CSE-BD Sec-I2

Aim: To Implement block world problem using Python

Code:

```python
import re
from itertools import permutations
from collections import deque
from sys import argv, exit
from time import perf_counter
from copy import deepcopy


def openProblem():

    objects = ['E', 'G', 'C', 'D', 'F', 'A', 'B']
    initTemp = [('CLEAR', 'B', ''), ('CLEAR', 'A', ''), ('ONTABLE', 'F', ''), ('ONTABLE', 'D', ''),
                ('ON', 'B', 'C'), ('ON', 'C', 'G'), ('ON', 'G', 'E'), ('ON', 'E', 'F'), ('ON', 'A', 'D')]
    goalTemp = [('E', 'B'), ('B', 'F'), ('F', 'D'),
                ('D', 'A'), ('A', 'C'), ('C', 'G')]

    init = {i: ['table', True] for i in objects}

    print('Init Temp', initTemp)
    print('Goal Temp', goalTemp)
    print('objects', objects)

    # print()

    for item in initTemp:
        if item[0] == 'ON':
            init[item[1]][0] = item[2]
            init[item[2]][1] = False
        # unessecary, but left for readability
        # elif item[0] == 'ONTABLE':
        #    state[item[1]][0] = 'table'
        # elif item[0] == 'CLEAR':
        #    state[item[1]][1] = True
        #####################################

    # Initialize goal and their state (position, is clear).
    goal = {i: ['table', True] for i in objects}
```

```python
    # For each item that's is on another, change it's location
    # and set to unclear.
    for item in goalTemp:
        goal[item[0]][0] = item[1]
        goal[item[1]][1] = False

    return init, goal, objects


def writeSolution(solution):
    print('\n')
    i = 0
    for move in solution:
        i += 1
        print('{}. Move({}, {}, {})\n' .format(
            i, move[0], move[1], move[2]))


class State(object):
    """
        description
            A state's description dictionary looks like this...
            {'A': ['B', True], 'C': ['table', True], 'B': ['table', False]}

            And represents...
            'That cube': ['is on top of that', is it clear on top?]

            And if we visualize it, it looks like this...
             ___
            | A |
            |___|  ___
            | B | | C |
            |___| |___|
            ====================== <-- table

        parent
            The parent state object.

        move
            The move that was required to form that state from parent state.
            The list has the following format...
            ['A', 'B', 'C'] or ['A', 'B', 'table']
            ...which means, move cube A, from cube, on top of cube C or table.
    """
```

```python
def __init__(self, description=None, parent=None, move=None):
    super(State, self).__init__()
    self._parent = parent
    self._moveToForm = move
    if not description:
        self._stateDescription = deepcopy(self._parent._stateDescription)
        # If that move doesn't exists, it probably means, that it's a root state.
        if self._moveToForm is not None:
            self.__move(self._moveToForm[0], self._moveToForm[2])
    # Otherwise, just use the state given as argument.
    else:
        self._stateDescription = description
def __eq__(self, other):
    if other is None:
        return False
    return self._stateDescription == other._stateDescription

# Overriding the representation method, for debugging purposes.
def __repr__(self):
    return str(self._stateDescription) + '\n'

def _generateStateChildren(self):
    """
        Generates all possible children (states) of itself (state).
        Each child state represents a possible move.
    """
    # Find all clear cubes of the state.
    clearCubes = [
        key for key in self._stateDescription if self._stateDescription[key][1] is True]
    possibleMoves = list(permutations(clearCubes, 2)) + [(
        cube, 'table') for cube in clearCubes if self._stateDescription[cube][0] != 'table']
    states = []
    for cubeToMove, destinationCube in possibleMoves:
        states.append(State(parent=self, move=self.__move(
            cubeToMove, destinationCube, True)))
    return states
def __move(self, object, destination, fake=False):
    """
        Moves the selected object to desired destination and
        returns the action in detail. Optionally,
        it only returns the hypothetical move, without actually doing it.
    """
    oldPosition = self._stateDescription[object][0]
    if fake:
```

```python
        if fake:
            return [object, oldPosition, destination]
        if oldPosition != 'table':
            self._stateDescription[oldPosition][1] = True
        self._stateDescription[object][0] = destination
        if destination != 'table':
            self._stateDescription[destination][1] = False
        move = [object, oldPosition, destination]
        return move
    def __hash__(self):
        string = ''
        for key, value in self._stateDescription.items():
            string += "".join(key + value[0] + str(value[1])[0])
        return hash(string)
    def _tracePath(self):
        """
            Finds the moves required to solve the problem.
        """
        path = []
        currentParent = self
        while currentParent._parent is not None:
            path.append(currentParent._moveToForm)
            currentParent = currentParent._parent
        return path[::-1]

    def _tracePathDEBUG(self):
        # Just pretty printing the moves to solution.
        i = 0
        for move in self._tracePath():
            i += 1
            print('{}. Move({}, {}, {})' .format(i, move[0], move[1], move[2]))
def breadthFirstSearch(initialState, goalState, timeout=60):
    # Initialize iterations counter.
    iterations = 0
    visited, queue = set(), deque([initialState])
    t1 = perf_counter()
    while queue:
        t2 = perf_counter()
        if t2 - t1 > timeout:
            return None, iterations
        iterations += 1
        vertex = queue.popleft()
        if vertex == goalState:
            return vertex._tracePath(), iterations
```

```python
        for neighbour in vertex._generateStateChildren():
            if neighbour not in visited:
                visited.add(neighbour)
                queue.append(neighbour)
def depthFirstSearch(initialState, goalState, timeout=60):
    iterations = 0
    visited, stack = set(), deque([initialState])
    t1 = perf_counter()
    while stack:
        t2 = perf_counter()
        if t2 - t1 > timeout:
            return None, iterations
        iterations += 1
        vertex = stack.pop()  # right
        if vertex == goalState:
            return vertex._tracePath(), iterations

        if vertex in visited:
            continue
        for neighbour in vertex._generateStateChildren():
            stack.append(neighbour)
        visited.add(vertex)
def __distanceFromGoal(currentStates, goalState):
    """ The H function. """
    statesScores = []
    for state in currentStates:
        outOfPlaceBlocks = 0
        for block in state._stateDescription:
            if state._stateDescription[block] != goalState._stateDescription[block]:
                outOfPlaceBlocks += 1
        statesScores.append(outOfPlaceBlocks)
    # Return the index of the state with smallest distance from goal.
    return statesScores.index(min(statesScores))
def __distanceFromGoalWithLeastMoves(currentStates, goalState):
    """ The G + H function. """
    statesScores = []
    for state in currentStates:
        outOfPlaceBlocks = 0
        for block in state._stateDescription:
            if state._stateDescription[block] != goalState._stateDescription[block]:
                outOfPlaceBlocks += 1
        statesScores.append(outOfPlaceBlocks + len(state._tracePath()))
    return statesScores.index(min(statesScores))
```

```python
def heuristicSearch(initialState, goalState, algorithm='best', timeout=60):
    if algorithm == 'astar':
        function = __distanceFromGoalWithLeastMoves
    elif algorithm == 'best':
        function = __distanceFromGoal
    iterations = 0
    visited, list = set(), [initialState]
    # Initialize timeout counter.
    t1 = perf_counter()
    while list:
        t2 = perf_counter()
        if t2 - t1 > timeout:
            return None, iterations
        iterations += 1
        item = function(list, goalState)
        vertex = list.pop(item)
        if vertex == goalState:
            return vertex._tracePath(), iterations
        for neighbour in vertex._generateStateChildren():
            if neighbour in visited:
                continue
            visited.add(neighbour)
            list.append(neighbour)

def main(argv):
    init, goal, cubes = openProblem()
    # Initialize inital and goal states.
    initialState = State(init)
    goalState = State(goal)
    algorithm = 'astar'
    t1 = perf_counter()
    if algorithm == 'breadth':
        solution, iters = breadthFirstSearch(initialState, goalState)
    elif algorithm == 'depth':
        solution, iters = depthFirstSearch(initialState, goalState)
    elif algorithm == 'best' or algorithm == 'astar':
        solution, iters = heuristicSearch(initialState, goalState, algorithm)
    else:
        raise Exception(
            'Unknown algorithm. Available : breadth, depth, best, astar')
    t2 = perf_counter()
    # print('| Problem name: {}' .format(' ' * 10 + problemFile))
    print('| Algorithm used: {}' .format(' ' * 8 + algorithm))
    print('| Number of cubes: {}' .format(' ' * 7 + str(len(cubes))))
```

```python
    print('| Cubes: {}' .format(' ' * 17 + str(' '.join(cubes))))
    if solution:
        print('| Solved in: {}' .format(' ' * 13 + str(t2-t1)))
        print('| Algorithm iterations: {}' .format(' ' * 2 + str(iters)))
        print('| Moves: {}' .format(' ' * 17 + str(len(solution))))

        print('| Solution:' + ' ' * 15 + 'Found!')
        writeSolution(solution)
    else:
        print('| Solution:' + ' ' * 15 + 'NOT found, search timed out.')


if __name__ == '__main__':
    main(argv)
```

Output:

```
Init Temp [('CLEAR', 'B', ''), ('CLEAR', 'A', ''), ('ONTABLE', 'F', ''), ('ONTABLE', 'D', ''), ('ON', 'B', 'C'), ('ON', 'C', 'G'), ('ON', 'G', 'E'), ('ON', 'E', 'F'), ('ON', 'A', 'D')]
Goal Temp [('E', 'B'), ('B', 'F'), ('F', 'D'), ('D', 'A'), ('A', 'C'), ('C', 'G')]
objects ['E', 'G', 'C', 'D', 'F', 'A', 'B']
| Algorithm used:          astar
| Number of cubes:         7
| Cubes:                   E G C D F A B
| Solved in:               40.36059043000023
| Algorithm iterations:    2595
| Moves:                   11
| Solution:                Found!


1. Move(B, C, table)

2. Move(C, G, A)

3. Move(G, E, table)

4. Move(E, F, B)

5. Move(C, A, G)

6. Move(A, D, C)

7. Move(D, table, A)

8. Move(F, table, D)

9. Move(E, B, table)

10. Move(B, table, F)

11. Move(E, table, B)
```

Result: We have successfully implemented the block world problem.