UDACITY

PROJECT

Implement a Planning Search

A part of the Artificial Intelligence Nanodegree and Specializations Program

PROJECT REVIEW

CODE REVIEW 2

NOTES

my_air_cargo_problems.py 1

```
from aimacode.logic import PropKB
from aimacode.planning import Action
from aimacode.search import (
    Node, Problem,
)
from aimacode.utils import expr
from lp_utils import (
    FluentState, encode_state, decode_state,
)
from my_planning_graph import PlanningGraph

from functools import lru_cache


class AirCargoProblem(Problem):
    def __init__(self, cargos, planes, airports, initial: FluentState, goal: list):
        """

        :param cargos: list of str
            cargos in the problem
        :param planes: list of str
            planes in the problem
        :param airports: list of str
            airports in the problem
        :param initial: FluentState object
            positive and negative literal fluents (as expr) describing initial state
        :param goal: list of expr
            literal fluents required for goal test
        """
        self.state_map = initial.pos + initial.neg
        self.initial_state_TF = encode_state(initial, self.state_map)
        Problem.__init__(self, self.initial_state_TF, goal=goal)
        self.cargos = cargos
        self.planes = planes
        self.airports = airports
        self.actions_list = self.get_actions()

    def get_actions(self):
        """
        This method creates concrete actions (no variables) for all actions in the problem
        domain action schema and turns them into complete Action objects as defined in the
        aimacode.planning module. It is computationally expensive to call this method directly;
        however, it is called in the constructor and the results cached in the `actions_list` property.

        Returns:
        ----------
        list<Action>
            list of Action objects
        """

        # TODO create concrete Action objects based on the domain action schema for: Load, Unload, and Fly
        # concrete actions definition: specific literal action that does not include variables as with the schema
        # for example, the action schema 'Load(c, p, a)' can represent the concrete actions 'Load(C1, P1, SFO)'
        # or 'Load(C2, P2, JFK)'.  The actions for the planning problem must be concrete because the problems in
        # forward search and Planning Graphs must use Propositional Logic

        def load_actions():
            """Create all concrete Load actions and return a list

            :return: list of Action objects
            """
            loads = []
            # TODO create all load ground actions from the domain Load action
            for c in self.cargos:
                for p in self.planes:
```

```
                    for a in self.airports:
                        precond_pos = [expr("At({}, {})".format(c, a)), expr("At({}, {})".format(p, a))]
                        precond_neg = []
                        effect_add = [expr("In({}, {})".format(c, p))]
                        effect_rem = [expr("At({}, {})".format(c, a))]
                        load = Action(expr("Load({}, {}, {})".format(c,p,a)),
                                      [precond_pos, precond_neg],
                                      [effect_add, effect_rem])
                        loads.append(load)
            return loads

        def unload_actions():
            """Create all concrete Unload actions and return a list

            :return: list of Action objects
            """
            unloads = []
            # TODO create all Unload ground actions from the domain Unload action
            for c in self.cargos:
                for p in self.planes:
                    for a in self.airports:
                        precond_pos = [expr("In({}, {})".format(c, p)), expr("At({}, {})".format(p, a))]
                        precond_neg = []
                        effect_add = [expr("At({}, {})".format(c, a))]
                        effect_rem = [expr("In({}, {})".format(c, p))]
                        unload = Action(expr("Unload({}, {}, {})".format(c, p, a)),
                                        [precond_pos, precond_neg],
                                        [effect_add, effect_rem])
                        unloads.append(unload)
            return unloads

        def fly_actions():
            """Create all concrete Fly actions and return a list

            :return: list of Action objects
            """
            flys = []
            for fr in self.airports:
                for to in self.airports:
                    if fr != to:
                        for p in self.planes:
                            precond_pos = [expr("At({}, {})".format(p, fr)),
                                           ]
                            precond_neg = []
                            effect_add = [expr("At({}, {})".format(p, to))]
                            effect_rem = [expr("At({}, {})".format(p, fr))]
                            fly = Action(expr("Fly({}, {}, {})".format(p, fr, to)),
                                         [precond_pos, precond_neg],
                                         [effect_add, effect_rem])
                            flys.append(fly)
            return flys

        return load_actions() + unload_actions() + fly_actions()

    def actions(self, state: str) -> list:
        """ Return the actions that can be executed in the given state.

        :param state: str
            state represented as T/F string of mapped fluents (state variables)
            e.g. 'FTTTFF'
        :return: list of Action objects
        """
        # TODO implement
        #create knowledge base
        kb = PropKB()
        #decode string of T/F as fluent per mapping
        FluentState = decode_state(state, self.state_map)
        #Add clauses of FluentState's positive sentence to the KB
        kb.tell(FluentState.pos_sentence())
        '''
        #one version to complete this function is to return the variable in the folloing line
        possible_actions = [ a for a in self.actions_list if a.check_precond(kb, a.args) ]
        '''
        #however, we use this longer version that run faster in our experiments
        #probably because it didn't call the action method check_precond from aimacode.planning
        possible_actions = []
        for action in self.actions_list:
            executable = True
            for clause in action.precond_pos:
                if clause not in kb.clauses:
                    executable = False
            for clause in action.precond_neg:
                if clause in kb.clauses:
                    executable = False
            if executable:
                possible_actions.append(action)

        return possible_actions

    def result(self, state: str, action: Action):
        """ Return the state that results from executing the given
        action in the given state. The action must be one of
        self.actions(state).

        :param state: state entering node
        :param action: Action applied
        :return: resulting state after action
        """
```

```
        # TODO implement
        new_state = FluentState([], [])
        old_state = decode_state(state, self.state_map)
        #add-list: 1. unaffected positive literals from old state
        for fluent in old_state.pos:
            if fluent not in action.effect_rem:
                new_state.pos.append(fluent)
        #add-list: 2. positive literals in the action's effect
        for fluent in action.effect_add:
            if fluent not in new_state.pos:
                new_state.pos.append(fluent)
        #delete-list: 1. unaffected negative literals from old state
        for fluent in old_state.neg:
            if fluent not in action.effect_add:
                new_state.neg.append(fluent)
        #delete-list: 2. negative literals in the action's effect
        for fluent in action.effect_rem:
            if fluent not in new_state.neg:
                new_state.neg.append(fluent)
        return encode_state(new_state, self.state_map)

    def goal_test(self, state: str) -> bool:
        """ Test the state to see if goal is reached

        :param state: str representing state
        :return: bool
        """
        kb = PropKB()
        kb.tell(decode_state(state, self.state_map).pos_sentence())
        for clause in self.goal:
            if clause not in kb.clauses:
                return False
        return True

    def h_1(self, node: Node):
        # note that this is not a true heuristic
        h_const = 1
        return h_const

    @lru_cache(maxsize=8192)
    def h_pg_levelsum(self, node: Node):
        """This heuristic uses a planning graph representation of the problem
        state space to estimate the sum of all actions that must be carried
        out from the current state in order to satisfy each individual goal
        condition.
        """
        # requires implemented PlanningGraph class
        pg = PlanningGraph(self, node.state)
        pg_levelsum = pg.h_levelsum()
        return pg_levelsum

    @lru_cache(maxsize=8192)
    def h_ignore_preconditions(self, node: Node):
        """This heuristic estimates the minimum number of actions that must be
        carried out from the current state in order to satisfy all of the goal
        conditions by ignoring the preconditions required for an action to be
        executed.
        """
        # TODO implement (see Russell-Norvig Ed-3 10.2.3  or Russell-Norvig Ed-2 11.2)
        count = 0
        kb = PropKB()
        #decode string of T/F as fluent per mapping
        FluentState = decode_state(node.state, self.state_map)
        #Add clauses of FluentState's positive sentence to the KB
        kb.tell(FluentState.pos_sentence())
        #for clause in kb.clauses:

        for clause in self.goal:
            if clause not in kb.clauses:
                #sometimes it reqire more that one action to reach the goal, but under-estimating is admissible
                #by this (inaccurate) counting, this heuristic is sometimes called number-of-unsatisfied-goals heuristic
                count += 1
        return count


def air_cargo_p1() -> AirCargoProblem:
    cargos = ['C1', 'C2']
    planes = ['P1', 'P2']
    airports = ['JFK', 'SFO']
    pos = [expr('At(C1, SFO)'),
           expr('At(C2, JFK)'),
           expr('At(P1, SFO)'),
           expr('At(P2, JFK)'),
           ]
    neg = [expr('At(C2, SFO)'),
           expr('In(C2, P1)'),
           expr('In(C2, P2)'),
           expr('At(C1, JFK)'),
           expr('In(C1, P1)'),
           expr('In(C1, P2)'),
           expr('At(P1, JFK)'),
           expr('At(P2, SFO)'),
           ]
    init = FluentState(pos, neg)
    goal = [expr('At(C1, JFK)'),
            expr('At(C2, SFO)'),
            ]
    return AirCargoProblem(cargos, planes, airports, init, goal)
```

```python


def air_cargo_p2() -> AirCargoProblem:
    # TODO implement Problem 2 definition
    cargos = ['C1', 'C2', 'C3']
    planes = ['P1', 'P2', 'P3']
    airports = ['JFK', 'SFO', 'ATL']
    #positive fluents
    pos = [expr('At(C1, SFO)'),
           expr('At(C2, JFK)'),
           expr('At(C3, ATL)'),
           expr('At(P1, SFO)'),
           expr('At(P2, JFK)'),
           expr('At(P3, ATL)'),
           ]
    #negative fluents
    neg = []
    for c in cargos:
        for a in airports:
            expression = expr('At({}, {})'.format(c, a))
            if expression not in pos:
                neg.append(expression)
    for p in planes:
        for a in airports:
            expression = expr('At({}, {})'.format(p, a))
            if expression not in pos:
                neg.append(expression)
    #none of the cargo had been loaded
    for c in cargos:
        for p in planes:
            neg.append(expr('In({}, {})'.format(c, p)))
    init = FluentState(pos, neg)
    goal = [expr('At(C1, JFK)'),
            expr('At(C2, SFO)'),
            expr('At(C3, SFO)'),
            ]
    print('init=', init)
    print('init.sentence=', init.sentence())
    print('init.pos_sentence=', init.pos_sentence())
    print('init.pos=', init.pos)
    print('goal=', goal)

    return AirCargoProblem(cargos, planes, airports, init, goal)


def air_cargo_p3() -> AirCargoProblem:
    # TODO implement Problem 3 definition
    cargos = ['C1', 'C2', 'C3', 'C4']
    planes = ['P1', 'P2']
    airports = ['JFK', 'SFO', 'ATL', 'ORD']
    #positive fluents
    pos = [expr('At(C1, SFO)'),
           expr('At(C2, JFK)'),
           expr('At(C3, ATL)'),
           expr('At(C4, ORD)'),
           expr('At(P1, SFO)'),
           expr('At(P2, JFK)'),
           ]
    #negative fluents
    neg = []
    for c in cargos:
        for a in airports:
            expression = expr('At({}, {})'.format(c, a))
            if expression not in pos:
                neg.append(expression)
    for p in planes:
        for a in airports:
            expression = expr('At({}, {})'.format(p, a))
            if expression not in pos:
                neg.append(expression)
    #none of the cargo had been loaded
    for c in cargos:
        for p in planes:
            neg.append(expr('In({}, {})'.format(c, p)))
    init = FluentState(pos, neg)
    goal = [expr('At(C1, JFK)'),
            expr('At(C2, SFO)'),
            expr('At(C3, JFK)'),
            expr('At(C4, SFO)'),
            ]
    return AirCargoProblem(cargos, planes, airports, init, goal)


```

AWESOME

Good work on implementing all the problem definitions.

▸ my_planning_graph.py 1

RETURN TO PATH

Rate this review

Student FAQ