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

▾ my_planning_graph.py 1

```
from aimacode.planning import Action
from aimacode.search import Problem
from aimacode.utils import expr
from lp_utils import decode_state


class PgNode():
    """Base class for planning graph nodes.

    includes instance sets common to both types of nodes used in a planning graph
    parents: the set of nodes in the previous level
    children: the set of nodes in the subsequent level
    mutex: the set of sibling nodes that are mutually exclusive with this node
    """

    def __init__(self):
        self.parents = set()
        self.children = set()
        self.mutex = set()

    def is_mutex(self, other) -> bool:
        """Boolean test for mutual exclusion

        :param other: PgNode
            the other node to compare with
        :return: bool
            True if this node and the other are marked mutually exclusive (mutex)
        """
        if other in self.mutex:
            return True
        return False

    def show(self):
        """helper print for debugging shows counts of parents, children, siblings

        :return:
            print only
        """
        print("{} parents".format(len(self.parents)))
        print("{} children".format(len(self.children)))
        print("{} mutex".format(len(self.mutex)))


class PgNode_s(PgNode):
    """A planning graph node representing a state (literal fluent) from a
    planning problem.

    Args:
    ----------
    symbol : str
        A string representing a literal expression from a planning problem
        domain.

    is_pos : bool
        Boolean flag indicating whether the literal expression is positive or
        negative.
    """

    def __init__(self, symbol: str, is_pos: bool):
        """S-level Planning Graph node constructor

        :param symbol: expr
```

```
        :param is_pos: bool
        Instance variables calculated:
            literal: expr
                    fluent in its literal form including negative operator if applicable
        Instance variables inherited from PgNode:
            parents: set of nodes connected to this node in previous A level; initially empty
            children: set of nodes connected to this node in next A level; initially empty
            mutex: set of sibling S-nodes that this node has mutual exclusion with; initially empty
        """
        PgNode.__init__(self)
        self.symbol = symbol
        self.is_pos = is_pos
        self.__hash = None

        self.literal = self.evaluate_literal()

    def evaluate_literal(self):
        literal = expr(self.symbol)
        if not self.is_pos:
            literal = '~' + literal
        return literal

    def show(self):
        """helper print for debugging shows literal plus counts of parents,
        children, siblings

        :return:
            print only
        """
        if self.is_pos:
            print("\n*** {}".format(self.symbol))
        else:
            print("\n*** ~{}".format(self.symbol))
        PgNode.show(self)

    def __eq__(self, other):
        """equality test for nodes - compares only the literal for equality

        :param other: PgNode_s
        :return: bool
        """
        return (isinstance(other, self.__class__) and
                self.is_pos == other.is_pos and
                self.symbol == other.symbol)

    def __hash__(self):
        self.__hash = self.__hash or hash(self.symbol) ^ hash(self.is_pos)
        return self.__hash


class PgNode_a(PgNode):
    """A-type (action) Planning Graph node - inherited from PgNode """


    def __init__(self, action: Action):
        """A-level Planning Graph node constructor

        :param action: Action
            a ground action, i.e. this action cannot contain any variables
        Instance variables calculated:
            An A-level will always have an S-level as its parent and an S-level as its child.
            The preconditions and effects will become the parents and children of the A-level node
            However, when this node is created, it is not yet connected to the graph
            prenodes: set of *possible* parent S-nodes
            effnodes: set of *possible* child S-nodes
            is_persistent: bool   True if this is a persistence action, i.e. a no-op action
        Instance variables inherited from PgNode:
            parents: set of nodes connected to this node in previous S level; initially empty
            children: set of nodes connected to this node in next S level; initially empty
            mutex: set of sibling A-nodes that this node has mutual exclusion with; initially empty
        """
        PgNode.__init__(self)
        self.action = action
        self.prenodes = self.precond_s_nodes()
        self.effnodes = self.effect_s_nodes()
        self.is_persistent = self.prenodes == self.effnodes
        self.__hash = None

    def show(self):
        """helper print for debugging shows action plus counts of parents, children, siblings

        :return:
            print only
        """
        print("\n*** {!s}".format(self.action))
        PgNode.show(self)

    def precond_s_nodes(self):
        """precondition literals as S-nodes (represents possible parents for this node).
        It is computationally expensive to call this function; it is only called by the
        class constructor to populate the `prenodes` attribute.

        :return: set of PgNode_s
        """
        nodes = set()
        for p in self.action.precond_pos:
            nodes.add(PgNode_s(p, True))
        for p in self.action.precond_neg:
```

```
            nodes.add(PgNode_s(p, False))
        return nodes

    def effect_s_nodes(self):
        """effect literals as S-nodes (represents possible children for this node).
        It is computationally expensive to call this function; it is only called by the
        class constructor to populate the `effnodes` attribute.

        :return: set of PgNode_s
        """
        nodes = set()
        for e in self.action.effect_add:
            nodes.add(PgNode_s(e, True))
        for e in self.action.effect_rem:
            nodes.add(PgNode_s(e, False))
        return nodes

    def __eq__(self, other):
        """equality test for nodes - compares only the action name for equality

        :param other: PgNode_a
        :return: bool
        """
        return (isinstance(other, self.__class__) and
                self.is_persistent == other.is_persistent and
                self.action.name == other.action.name and
                self.action.args == other.action.args)

    def __hash__(self):
        self.__hash = self.__hash or hash(self.action.name) ^ hash(self.action.args)
        return self.__hash


def mutexify(node1: PgNode, node2: PgNode):
    """ adds sibling nodes to each other's mutual exclusion (mutex) set. These should be sibling nodes!

    :param node1: PgNode (or inherited PgNode_a, PgNode_s types)
    :param node2: PgNode (or inherited PgNode_a, PgNode_s types)
    :return:
        node mutex sets modified
    """
    if type(node1) != type(node2):
        raise TypeError('Attempted to mutex two nodes of different types')
    node1.mutex.add(node2)
    node2.mutex.add(node1)


class PlanningGraph():
    """
    A planning graph as described in chapter 10 of the AIMA text. The planning
    graph can be used to reason about
    """

    def __init__(self, problem: Problem, state: str, serial_planning=True):
        """
        :param problem: PlanningProblem (or subclass such as AirCargoProblem or HaveCakeProblem)
        :param state: str (will be in form TFTTFF... representing fluent states)
        :param serial_planning: bool (whether or not to assume that only one action can occur at a time)
        Instance variable calculated:
            fs: FluentState
                the state represented as positive and negative fluent literal lists
            all_actions: list of the PlanningProblem valid ground actions combined with calculated no-op actions
            s_levels: list of sets of PgNode_s, where each set in the list represents an S-level in the planning graph
            a_levels: list of sets of PgNode_a, where each set in the list represents an A-level in the planning graph
        """
        self.problem = problem
        self.fs = decode_state(state, problem.state_map)
        self.serial = serial_planning
        self.all_actions = self.problem.actions_list + self.noop_actions(self.problem.state_map)
        self.s_levels = []
        self.a_levels = []
        self.create_graph()

    def noop_actions(self, literal_list):
        """create persistent action for each possible fluent

        "No-Op" actions are virtual actions (i.e., actions that only exist in
        the planning graph, not in the planning problem domain) that operate
        on each fluent (literal expression) from the problem domain. No op
        actions "pass through" the literal expressions from one level of the
        planning graph to the next.

        The no-op action list requires both a positive and a negative action
        for each literal expression. Positive no-op actions require the literal
        as a positive precondition and add the literal expression as an effect
        in the output, and negative no-op actions require the literal as a
        negative precondition and remove the literal expression as an effect in
        the output.

        This function should only be called by the class constructor.

        :param literal_list:
        :return: list of Action
        """
        action_list = []
        for fluent in literal_list:
            act1 = Action(expr("Noop_pos({})".format(fluent)), ([fluent], []), ([fluent], []))
            action_list.append(act1)
```

```
            act2 = Action(expr("Noop_neg({})".format(fluent)), ([], [fluent]), ([], [fluent]))
            action_list.append(act2)
        return action_list

    def create_graph(self):
        """ build a Planning Graph as described in Russell-Norvig 3rd Ed 10.3 or 2nd Ed 11.4

        The S0 initial level has been implemented for you.  It has no parents and includes all of
        the literal fluents that are part of the initial state passed to the constructor.  At the start
        of a problem planning search, this will be the same as the initial state of the problem.  However,
        the planning graph can be built from any state in the Planning Problem

        This function should only be called by the class constructor.

        :return:
            builds the graph by filling s_levels[] and a_levels[] lists with node sets for each level
        """
        # the graph should only be built during class construction
        if (len(self.s_levels) != 0) or (len(self.a_levels) != 0):
            raise Exception(
                'Planning Graph already created; construct a new planning graph for each new state in the planning sequence')

        # initialize S0 to literals in initial state provided.
        leveled = False
        level = 0
        self.s_levels.append(set())  # S0 set of s_nodes - empty to start
        # for each fluent in the initial state, add the correct literal PgNode_s
        for literal in self.fs.pos:
            self.s_levels[level].add(PgNode_s(literal, True))
        for literal in self.fs.neg:
            self.s_levels[level].add(PgNode_s(literal, False))
        # no mutexes at the first level

        # continue to build the graph alternating A, S levels until last two S levels contain the same literals,
        # i.e. until it is "leveled"
        while not leveled:
            self.add_action_level(level)
            self.update_a_mutex(self.a_levels[level])

            level += 1
            self.add_literal_level(level)
            self.update_s_mutex(self.s_levels[level])

            if self.s_levels[level] == self.s_levels[level - 1]:
                leveled = True

    def add_action_level(self, level):
        """ add an A (action) level to the Planning Graph

        :param level: int
            the level number alternates S0, A0, S1, A1, S2, .... etc the level number is also used as the
            index for the node set lists self.a_levels[] and self.s_levels[]
        :return:
            adds A nodes to the current level in self.a_levels[level]
        """
        # TODO add action A level to the planning graph as described in the Russell-Norvig text
        # 1. determine what actions to add and create those PgNode_a objects
        # 2. connect the nodes to the previous S literal level
        # for example, the A0 level will iterate through all possible actions for the problem and add a PgNode_a to a_levels[0]
        #   set iff all prerequisite literals for the action hold in S0.  This can be accomplished by testing
        #   to see if a proposed PgNode_a has prenodes that are a subset of the previous S level.  Once an
        #   action node is added, it MUST be connected to the S node instances in the appropriate s_level set.

        #determine what actions to add and create those PgNode_a objects
        new_a_level = []
        for action in self.all_actions:
            a_node = PgNode_a(action)
            if a_node.prenodes.issubset(self.s_levels[level]):
                new_a_level.append(a_node)
        self.a_levels.append(new_a_level)

        #connect the nodes to the previous S literal level
        children = set(self.a_levels[level])
        for s_node in self.s_levels[level]:
            s_node.children.update(children)
        parents = set(self.s_levels[level])
        for a_node in self.a_levels[level]:
            a_node.parents.update(parents)

        '''
        #record prerequisite avaliable from previous S literal level
        pos_sentence = []
        neg_sentence = []
        for s_node in self.s_levels[level]:
            if s_node.is_pos:
                pos_sentence.append(s_node.symbol)
            else:
                neg_sentence.append(s_node.symbol)

        #determine what actions to add and create those PgNode_a objects
        new_a_level = []

        for action in self.all_actions:
            to_Add = True
            for literal in action.precond_pos:
                if literal not in pos_sentence:
                    to_Add = False
            for literal in action.precond_neg:
```

```
                if literal not in neg_sentence:
                    to_Add = False
            if to_Add:
                a_node = PgNode_a(action)
                new_a_level.append(a_node)

        self.a_levels.append(new_a_level)
        '''



        #for node in self.s_levels[level]:
        #    persistent_node =

    def add_literal_level(self, level):
```

AWESOME

Correctly implemented `add_literal_level` and `add_action_level`

```
        """ add an S (literal) level to the Planning Graph

        :param level: int
            the level number alternates S0, A0, S1, A1, S2, .... etc the level number is also used as the
            index for the node set lists self.a_levels[] and self.s_levels[]
        :return:
            adds S nodes to the current level in self.s_levels[level]
        """
        # TODO add literal S level to the planning graph as described in the Russell-Norvig text
        # 1. determine what literals to add
        # 2. connect the nodes
        # for example, every A node in the previous level has a list of S nodes in effnodes that represent the effect
        #   produced by the action.  These literals will all be part of the new S level.  Since we are working with sets, they
        #   may be "added" to the set without fear of duplication.  However, it is important to then correctly create and connect
        #   all of the new S nodes as children of all the A nodes that could produce them, and likewise add the A nodes to the
        #   parent sets of the S nodes

        #determine what literals to add
        new_s_level = set()
        for a_node in self.a_levels[level-1]:
            new_s_level.update(a_node.effect_s_nodes())
        self.s_levels.append(list(new_s_level))

        #connect the nodes to the previous S literal level
        children = set(self.s_levels[level])
        for a_node in self.a_levels[level-1]:
            a_node.children.update(children)
        parents = set(self.a_levels[level-1])
        for s_node in self.s_levels[level]:
            s_node.parents.update(parents)

    def update_a_mutex(self, nodeset):
        """ Determine and update sibling mutual exclusion for A-level nodes

        Mutex action tests section from 3rd Ed. 10.3 or 2nd Ed. 11.4
        A mutex relation holds between two actions a given level
        if the planning graph is a serial planning graph and the pair are nonpersistence actions
        or if any of the three conditions hold between the pair:
           Inconsistent Effects
           Interference
           Competing needs

        :param nodeset: set of PgNode_a (siblings in the same level)
        :return:
            mutex set in each PgNode_a in the set is appropriately updated
        """
        nodelist = list(nodeset)
        for i, n1 in enumerate(nodelist[:-1]):
            for n2 in nodelist[i + 1:]:
                if (self.serialize_actions(n1, n2) or
                        self.inconsistent_effects_mutex(n1, n2) or
                        self.interference_mutex(n1, n2) or
                        self.competing_needs_mutex(n1, n2)):
                    mutexify(n1, n2)

    def serialize_actions(self, node_a1: PgNode_a, node_a2: PgNode_a) -> bool:
        """
        Test a pair of actions for mutual exclusion, returning True if the
        planning graph is serial, and if either action is persistent; otherwise
        return False.  Two serial actions are mutually exclusive if they are
        both non-persistent.

        :param node_a1: PgNode_a
        :param node_a2: PgNode_a
        :return: bool
        """
        #
        if not self.serial:
            return False
        if node_a1.is_persistent or node_a2.is_persistent:
            return False
        return True

    def inconsistent_effects_mutex(self, node_a1: PgNode_a, node_a2: PgNode_a) -> bool:
        """
```

```
        Test a pair of actions for inconsistent effects, returning True if
        one action negates an effect of the other, and False otherwise.

        HINT: The Action instance associated with an action node is accessible
        through the PgNode_a.action attribute. See the Action class
        documentation for details on accessing the effects and preconditions of
        an action.

        :param node_a1: PgNode_a
        :param node_a2: PgNode_a
        :return: bool
        """
        # TODO test for Inconsistent Effects between nodes
        effect1 = list(node_a1.effnodes)
        effect2 = list(node_a2.effnodes)
        for e1 in effect1:
            for e2 in effect2:
                if self.negation_mutex(e1, e2):
                    return True
        return False

        '''
        # a more compact way of writing this function might be:

        return  ( len(list(set(node_a1.action.effect_add) & set(node_a2.action.precond_neg))) >0
                   or len(list(set(node_a2.action.effect_add) & set(node_a1.action.precond_neg))) >0
                      or len(list(set(node_a1.action.effect_rem) & set(node_a2.action.precond_pos))) >0
                          or len(list(set(node_a2.action.effect_rem) & set(node_a1.action.precond_pos))) > 0
                  )

        '''
    def interference_mutex(self, node_a1: PgNode_a, node_a2: PgNode_a) -> bool:
        """
        Test a pair of actions for mutual exclusion, returning True if the
        effect of one action is the negation of a precondition of the other.

        HINT: The Action instance associated with an action node is accessible
        through the PgNode_a.action attribute. See the Action class
        documentation for details on accessing the effects and preconditions of
        an action.

        :param node_a1: PgNode_a
        :param node_a2: PgNode_a
        :return: bool
        """
        # TODO test for Interference between nodes
        pre_con1 = list(node_a1.prenodes)
        pre_con2 = list(node_a2.prenodes)
        effect1 = list(node_a1.effnodes)
        effect2 = list(node_a2.effnodes)
        for e1 in effect1:
            for p2 in pre_con2:
                if self.negation_mutex(e1, p2):
                    return True
        for e2 in effect2:
            for p1 in pre_con1:
                if self.negation_mutex(e2, p1):
                    return True
        return False

    def competing_needs_mutex(self, node_a1: PgNode_a, node_a2: PgNode_a) -> bool:
        """
        Test a pair of actions for mutual exclusion, returning True if one of
        the precondition of one action is mutex with a precondition of the
        other action.

        :param node_a1: PgNode_a
        :param node_a2: PgNode_a
        :return: bool
        """

        # TODO test for Competing Needs between nodes
        pre_con1 = list(node_a1.parents)
        pre_con2 = list(node_a2.parents)
        for p1 in pre_con1:
            for p2 in pre_con2:
                if p1 in p2.mutex:
                    return True
        return False

    def update_s_mutex(self, nodeset: set):
        """ Determine and update sibling mutual exclusion for S-level nodes

        Mutex action tests section from 3rd Ed. 10.3 or 2nd Ed. 11.4
        A mutex relation holds between literals at a given level
        if either of the two conditions hold between the pair:
           Negation
           Inconsistent support

        :param nodeset: set of PgNode_a (siblings in the same level)
        :return:
            mutex set in each PgNode_a in the set is appropriately updated
        """
        nodelist = list(nodeset)
        for i, n1 in enumerate(nodelist[:-1]):
            for n2 in nodelist[i + 1:]:
                if self.negation_mutex(n1, n2) or self.inconsistent_support_mutex(n1, n2):
                    mutexify(n1, n2)
```

```python
    def negation_mutex(self, node_s1: PgNode_s, node_s2: PgNode_s) -> bool:
        """
        Test a pair of state literals for mutual exclusion, returning True if
        one node is the negation of the other, and False otherwise.

        HINT: Look at the PgNode_s.__eq__ defines the notion of equivalence for
        literal expression nodes, and the class tracks whether the literal is
        positive or negative.

        :param node_s1: PgNode_s
        :param node_s2: PgNode_s
        :return: bool
        """
        # TODO test for negation between nodes
        if node_s1.symbol == node_s2.symbol:
            if not node_s1.is_pos == node_s2.is_pos:
                return True
        return False

    def inconsistent_support_mutex(self, node_s1: PgNode_s, node_s2: PgNode_s):
        """
        Test a pair of state literals for mutual exclusion, returning True if
        there are no actions that could achieve the two literals at the same
        time, and False otherwise.  In other words, the two literal nodes are
        mutex if all of the actions that could achieve the first literal node
        are pairwise mutually exclusive with all of the actions that could
        achieve the second literal node.

        HINT: The PgNode.is_mutex method can be used to test whether two nodes
        are mutually exclusive.

        :param node_s1: PgNode_s
        :param node_s2: PgNode_s
        :return: bool
        """
        # TODO test for Inconsistent Support between nodes
        a_nodes_for_1 = list(node_s1.parents)
        a_nodes_for_2 = list(node_s2.parents)
        for a1 in a_nodes_for_1:
            for a2 in a_nodes_for_2:
                if not a1.is_mutex(a2):
                    return False
        return True

    def h_levelsum(self) -> int:
        """The sum of the level costs of the individual goals (admissible if goals independent)

        :return: int
        """
        level_sum = 0
        # TODO implement
        # for each goal in the problem, determine the level cost, then add them together
        for goal in self.problem.goal:
            found = False
            for level in range(len(self.s_levels)):
                for s_node in self.s_levels[level]:
                    #literal = expr(s_node.symbol)
                    #if not s_node.is_pos:
                    #    literal = '~' + literal
                    #if goal == literal:
                    if goal == s_node.literal:
                        level_sum += level
                        found = True
                        break
                if found:
                    break
        return level_sum

```

RETURN TO PATH

Rate this review

Student FAQ