Build a Forward Planning Agent

▾ my_planning_graph.py 4

```python
from itertools import chain, combinations
from aimacode.planning import Action
from aimacode.utils import expr

from layers import BaseActionLayer, BaseLiteralLayer, makeNoOp, make_node


class ActionLayer(BaseActionLayer):

    def _inconsistent_effects(self, actionA, actionB):
        """ Return True if an effect of one action negates an effect of the other

        Hints:
            (1) `~Literal` can be used to logically negate a literal
            (2) `self.children` contains a map from actions to effects

        See Also
        --------
        layers.ActionNode
        """
        # TODO: implement this function
        # raise NotImplementedError
        return (self.children[actionA] & set([~i for i in self.children[actionB]]))!=set()
```

AWESOME

great, you are the first person -I grade- to use this trick (and I always suggest in my review that they use it) .. good work

```python

    def _interference(self, actionA, actionB):
        """ Return True if the effects of either action negate the preconditions of the other

        Hints:
            (1) `~Literal` can be used to logically negate a literal
            (2) `self.parents` contains a map from actions to preconditions

        See Also
        --------
        layers.ActionNode
        """
        # TODO: implement this function
        # raise NotImplementedError
        return (self.parents[actionA] & set([~i for i in self.children[actionB]]))!=set() or (self.parents[actionB] & set(

    def _competing_needs(self, actionA, actionB):
        """ Return True if any preconditions of the two actions are pairwise mutex in the parent layer

        Hints:
            (1) `self.parent_layer` contains a reference to the previous literal layer
            (2) `self.parents` contains a map from actions to preconditions

        See Also
        --------
        layers.ActionNode
        layers.BaseLayer.parent_layer
        """
        # TODO: implement this function
        # raise NotImplementedError
        for action_a_pc in actionA.preconditions:
            for action_b_pc in actionB.preconditions:
                if self.parent_layer.is_mutex(action_a_pc, action_b_pc):
                    return True # cond are mutex in parent_layer
        return False

class LiteralLayer(BaseLiteralLayer):

    def _inconsistent_support(self, literalA, literalB):
        """ Return True if all ways to achieve both literals are pairwise mutex in the parent layer

        Hints:
            (1) `self.parent_layer` contains a reference to the previous action layer
            (2) `self.parents` contains a map from literals to actions in the parent layer

        See Also
        --------
        layers.BaseLayer.parent_layer
        """
        # TODO: implement this function
        # raise NotImplementedError
        for lit_a_act in self.parents[literalA]:
            for lit_b_act in self.parents[literalB]:
                if not self.parent_layer.is_mutex(lit_a_act, lit_b_act):
                    return False
```

AWESOME

well done

```python
        # if both literals are mutex in self.parent_layer
        return True


    def _negation(self, literalA, literalB):
        """ Return True if two literals are negations of each other """
        # TODO: implement this function
        # raise NotImplementedError
        if literalA == ~literalB:
            return True
        return False

class PlanningGraph:
    def __init__(self, problem, state, serialize=True, ignore_mutexes=False):
        """
        Parameters
        ----------
        problem : PlanningProblem
            An instance of the PlanningProblem class

        state : tuple(bool)
            An ordered sequence of True/False values indicating the literal value
```

```
            of the corresponding fluent in problem.state_map

        serialize : bool
            Flag indicating whether to serialize non-persistence actions. Actions
            should NOT be serialized for regression search (e.g., GraphPlan), and
            _should_ be serialized if the planning graph is being used to estimate
            a heuristic
        """
        self._serialize = serialize
        self._is_leveled = False
        self._ignore_mutexes = ignore_mutexes
        self.goal = set(problem.goal)

        # make no-op actions that persist every literal to the next layer
        no_ops = [make_node(n, no_op=True) for n in chain(*(makeNoOp(s) for s in problem.state_map))]
        self._actionNodes = no_ops + [make_node(a) for a in problem.actions_list]

        # initialize the planning graph by finding the literals that are in the
        # first layer and finding the actions they they should be connected to
        literals = [s if f else ~s for f, s in zip(state, problem.state_map)]
        layer = LiteralLayer(literals, ActionLayer(), self._ignore_mutexes)
        layer.update_mutexes()
        self.literal_layers = [layer]
        self.action_layers = []

    def find_level_costs(self): # find goals and layer costs
        level_num = 0
        costs = [0]
        goals_finished = set()

        for goals in self.goal:
            if goals in self.literal_layers[0]:
                goals_finished.add(goals)

        if not self.goal - goals_finished:
            return costs

        while not self._is_leveled:
            self._extend()
```

AWESOME

you nailed this part :D (y)

```
            level_num += 1

            for goal in self.goal - goals_finished:
                if goal in self.literal_layers[level_num]:
                    costs.append(level_num)
                    goals_finished.add(goal)

            if not self.goal - goals_finished:
                return costs

    def h_levelsum(self):
        """ Calculate the level sum heuristic for the planning graph

        The level sum is the sum of the level costs of all the goal literals
        combined. The "level cost" to achieve any single goal literal is the
        level at which the literal first appears in the planning graph. Note
        that the level cost is **NOT** the minimum number of actions to
        achieve a single goal literal.

        For example, if Goal_1 first appears in level 0 of the graph (i.e.,
        it is satisfied at the root of the planning graph) and Goal_2 first
        appears in level 3, then the levelsum is 0 + 3 = 3.

        Hints
        -----
          (1) See the pseudocode folder for help on a simple implementation
          (2) You can implement this function more efficiently than the
              sample pseudocode if you expand the graph one level at a time
              and accumulate the level cost of each goal rather than filling
              the whole graph at the start.

        See Also
        --------
        Russell-Norvig 10.3.1 (3rd Edition)
        """
        # TODO: implement this function
        # raise NotImplementedError
        return (sum(self.find_level_costs()))

    def h_maxlevel(self):
        """ Calculate the max level heuristic for the planning graph

        The max level is the largest level cost of any single goal fluent.
        The "level cost" to achieve any single goal literal is the level at
        which the literal first appears in the planning graph. Note that
        the level cost is **NOT** the minimum number of actions to achieve
        a single goal literal.

        For example, if Goal1 first appears in level 1 of the graph and
        Goal2 first appears in level 3, then the levelsum is max(1, 3) = 3.

        Hints
        -----
          (1) See the pseudocode folder for help on a simple implementation
          (2) You can implement this function more efficiently if you expand
              the graph one level at a time until the last goal is met rather
              than filling the whole graph at the start.

        See Also
        --------
        Russell-Norvig 10.3.1 (3rd Edition)

        Notes
        -----
        WARNING: you should expect long runtimes using this heuristic with A*
        """
        # TODO: implement maxlevel heuristic
        # raise NotImplementedError
        return (max(self.find_level_costs()))

    def h_setlevel(self):
        """ Calculate the set level heuristic for the planning graph

        The set level of a planning graph is the first level where all goals
        appear such that no pair of goal literals are mutex in the last
        layer of the planning graph.

        Hints
        -----
          (1) See the pseudocode folder for help on a simple implementation
          (2) You can implement this function more efficiently if you expand
              the graph one level at a time until you find the set level rather
              than filling the whole graph at the start.

        See Also
        --------
        Russell-Norvig 10.3.1 (3rd Edition)

        Notes
        -----
        WARNING: you should expect long runtimes using this heuristic on complex problems
        """
        # TODO: implement setlevel heuristic
        # raise NotImplementedError
        level_num = 0
        while not self._is_leveled:
```

SUGGESTION

there is a hidden (and common) mistake here which doesn't affect your results because we test each heuristic separably, there is a hidden assumption in your code that the only function modifying the state (self._is_leveled and self.literal_layers) is the current heuristic function. this is of course true for the way we are testing that the only function modifying the state (self._is_leveled and self.literal_layers) is the current heuristic function, this is of course true for the way we are testing

but in general you can expect that multiple heuristics might be used simultaneously and so you should take care of that (e.g. loop through self.literal_layers before entering this loop, that works in this case because it only checks layer 0 and in the general case it will check all layers that were expanded -including those expanded by other heuristics)

```
            finished_goals = True
            extended_layers = self.literal_layers[-1]

            for goals in self.goal:
                if goals not in extended_layers:
                    finished_goals = False

            if finished_goals == False:
                level_num += 1
                self._extend()
                continue

            mutex_all = False
            for a_goal in self.goal:
                for b_goal in self.goal:
                    if extended_layers.is_mutex(a_goal, b_goal):
                        mutex_all = True

            if mutex_all == False:
                return (level_num)

            level_num += 1
            self._extend()

        return (level_num)

    ##############################################################################
    #                     DO NOT MODIFY CODE BELOW THIS LINE                     #
    ##############################################################################

    def fill(self, maxlevels=-1):
        """ Extend the planning graph until it is leveled, or until a specified number of
        levels have been added

        Parameters
        ----------
        maxlevels : int
            The maximum number of levels to extend before breaking the loop. (Starting with
            a negative value will never interrupt the loop.)

        Notes
        -----
        YOU SHOULD NOT THIS FUNCTION TO COMPLETE THE PROJECT, BUT IT MAY BE USEFUL FOR TESTING
        """
        while not self._is_leveled:
            if maxlevels == 0: break
            self._extend()
            maxlevels -= 1
        return self

    def _extend(self):
        """ Extend the planning graph by adding both a new action layer and a new literal layer

        The new action layer contains all actions that could be taken given the positive AND
        negative literals in the leaf nodes of the parent literal level.

        The new literal layer contains all literals that could result from taking each possible
        action in the NEW action layer.
        """
        if self._is_leveled: return

        parent_literals = self.literal_layers[-1]
        parent_actions = parent_literals.parent_layer
        action_layer = ActionLayer(parent_actions, parent_literals, self._serialize, self._ignore_mutexes)
        literal_layer = LiteralLayer(parent_literals, action_layer, self._ignore_mutexes)

        for action in self._actionNodes:
            # actions in the parent layer are skipped because are added monotonically to planning graphs,
            # which is performed automatically in the ActionLayer and LiteralLayer constructors
            if action not in parent_actions and action.preconditions <= parent_literals:
                action_layer.add(action)
                literal_layer |= action.effects

                # add two-way edges in the graph connecting the parent layer with the new action
                parent_literals.add_outbound_edges(action, action.preconditions)
                action_layer.add_inbound_edges(action, action.preconditions)

                # # add two-way edges in the graph connecting the new literaly layer with the new action
                action_layer.add_outbound_edges(action, action.effects)
                literal_layer.add_inbound_edges(action, action.effects)

        action_layer.update_mutexes()
        literal_layer.update_mutexes()
        self.action_layers.append(action_layer)
        self.literal_layers.append(literal_layer)
        self._is_leveled = literal_layer == action_layer.parent_layer
```

RETURN TO PATH

Rate this review