‹ Return to "Artificial Intelligence Nanodegree" in the classroom

Build a Forward Planning Agent

REVIEW | CODE REVIEW 3 | ANNOTATIONS 1 | HISTORY

▾ my_planning_graph.py 3

```

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

from layers import BaseActionLayer, BaseLiteralLayer, makeNo


class ActionLayer(BaseActionLayer):

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

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

        See Also
        --------
        layers.ActionNode
        """
        return any([~e in actionA.effects for e in actionB.ef


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

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

```

```
        See Also
        --------
        layers.ActionNode
        """
        return any([~e in actionA.preconditions for e in acti


    def _competing_needs(self, actionA, actionB):
        """ Return True if any preconditions of the two actic

        Hints:
            (1) `self.parent_layer` contains a reference to t
            (2) `self.parents` contains a map from actions tc

        See Also
        --------
        layers.ActionNode
        layers.BaseLayer.parent_layer
        """
        return any(self.parent_layer.is_mutex(itemA, itemB) f


class LiteralLayer(BaseLiteralLayer):

    def _inconsistent_support(self, literalA, literalB):
        """ Return True if all ways to achieve both literals

        Hints:
            (1) `self.parent_layer` contains a reference to t
            (2) `self.parents` contains a map from literals t

        See Also
        --------
        layers.BaseLayer.parent_layer
        """
        return all(self.parent_layer.is_mutex(actionA, actior

    def _negation(self, literalA, literalB):
        """ Return True if two literals are negations of each
        return literalA == ~literalB or literalB == ~literalA


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

        state : tuple(bool)
            An ordered sequence of True/False values indicati
            of the corresponding fluent in problem.state_map

        serialize : bool
            Flag indicating whether to serialize non-persiste
            should NOT be serialized for regression search (e
            _should_ be serialized if the planning graph is b
            a heuristic
        """
```

```python
        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 th
        no_ops = [make_node(n, no_op=True) for n in chain(*(r
        self._actionNodes = no_ops + [make_node(a) for a in p

        # initialize the planning graph by finding the litera
        # first layer and finding the actions they they shoul
        literals = [s if f else ~s for f, s in zip(state, pro
        layer = LiteralLayer(literals, ActionLayer(), self._i
        layer.update_mutexes()
        self.literal_layers = [layer]
        self.action_layers = []

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

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

        For example, if Goal_1 first appears in level 0 of th
        it is satisfied at the root of the planning graph) ar
        appears in level 3, then the levelsum is 0 + 3 = 3.

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

        See Also
        --------
        Russell-Norvig 10.3.1 (3rd Edition)
        """
        self.fill()

        i = 0
        for g in self.goal:
            for idx,layer in enumerate(self.literal_layers):
                if g in layer:
                    i += idx
                    break
        return i
```

AWESOME

Excellent

```python
    def h_maxlevel(self):
        """ Calculate the max level heuristic for the plannin

        The max level is the largest level cost of any single
        The "level cost" to achieve any single goal literal i
        which the literal first appears in the planning graph
        the level cost is **NOT** the minimum number of actic
        a single goal literal.

        For example, if Goal1 first appears in level 1 of the
        Goal2 first appears in level 3, then the levelsum is

        Hints
        -----
          (1) See the pseudocode folder for help on a simple
          (2) You can implement this function more efficientl
              the graph one level at a time until the last gc
              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 h
        """
        # maxlevel heuristic
        self.fill()
        i = 0
        for g in self.goal:
            for idx,layer in enumerate(self.literal_layers):
                if g in layer:
                    i = max(i,idx)
                    break
        return i

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

        The set level of a planning graph is the first level
        appear such that no pair of goal literals are mutex i
        layer of the planning graph.

        Hints
        -----
          (1) See the pseudocode folder for help on a simple
          (2) You can implement this function more efficientl
              the graph one level at a time until you find th
              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 h
        """
        # setlevel heuristic
```

```
        def all_goals(layer):
            for g in self.goal:
                if g not in layer:
                    return False
            return True

        def no_mutex(layer):
            for g1, g2 in combinations(self.goal, 2):
                if layer.is_mutex(g1, g2):
                    return False
            return True

        level = 0
        while not self._is_leveled:
            layer = self.literal_layers[-1]
            if all_goals(layer) and no_mutex(layer):
                return level

            self._extend()
            level += 1
```

AWESOME

Brilliant.

```
        return -1


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

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

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

        Notes
        -----
        YOU SHOULD NOT THIS FUNCTION TO COMPLETE THE PROJECT,
        """
        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 ac

        The new action layer contains all actions that could
        negative literals in the leaf nodes of the parent lit
```

```
        The new literal layer contains all literals that coul
        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_lit
        literal_layer = LiteralLayer(parent_literals, action_

        for action in self._actionNodes:
            # actions in the parent layer are skipped becaus
            # which is performed automatically in the ActionI
            if action not in parent_actions and action.precon
                action_layer.add(action)
                literal_layer |= action.effects

                # add two-way edges in the graph connecting t
                parent_literals.add_outbound_edges(action, ac
                action_layer.add_inbound_edges(action, action

                # # add two-way edges in the graph connecting
                action_layer.add_outbound_edges(action, actio
                literal_layer.add_inbound_edges(action, actio

        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.pare
```

AWESOME

This code is really based on personally creativity and I can confidently say this proj creativity.. Keep up the good work.

RETURN TO PATH