

[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

```
1
2 from itertools import chain, combinations
3 from aimacode.planning import Action
4 from aimacode.utils import expr
5
6 from layers import BaseActionLayer, BaseLiteralLayer, makeNode
7
8
9 class ActionLayer(BaseActionLayer):
10
11     def _inconsistent_effects(self, actionA, actionB):
12         """ Return True if an effect of one action negates an effect of another
13
14         Hints:
15             (1) `~Literal` can be used to logically negate a literal
16             (2) `self.children` contains a map from actions to their effects
17
18         See Also
19         -----
20         layers.ActionNode
21         """
22         return any([~e in actionA.effects for e in actionB.effects])
23
24
25     def _interference(self, actionA, actionB):
26         """ Return True if the effects of either action negate the preconditions of the other
27
28         Hints:
29             (1) `~Literal` can be used to logically negate a literal
30             (2) `self.parents` contains a map from actions to their preconditions
31         """
```

```

32     See Also
33     -----
34     layers.ActionNode
35     """
36     return any([~e in actionA.preconditions for e in actionB.preconditions])
37
38
39     def _competing_needs(self, actionA, actionB):
40         """ Return True if any preconditions of the two actions are competing
41
42         Hints:
43             (1) `self.parent_layer` contains a reference to the layer
44             (2) `self.parents` contains a map from actions to layers
45
46         See Also
47         -----
48         layers.ActionNode
49         layers.BaseLayer.parent_layer
50         """
51         return any(self.parent_layer.is_mutex(itemA, itemB) for itemA in actionA.preconditions, itemB in actionB.preconditions)
52
53
54     class LiteralLayer(BaseLiteralLayer):
55
56         def _inconsistent_support(self, literalA, literalB):
57             """ Return True if all ways to achieve both literals are inconsistent
58
59             Hints:
60                 (1) `self.parent_layer` contains a reference to the layer
61                 (2) `self.parents` contains a map from literals to layers
62
63             See Also
64             -----
65             layers.BaseLayer.parent_layer
66             """
67             return all(self.parent_layer.is_mutex(actionA, actionB) for actionA in literalA.actions, actionB in literalB.actions)
68
69         def _negation(self, literalA, literalB):
70             """ Return True if two literals are negations of each other
71             return literalA == ~literalB or literalB == ~literalA
72
73
74     class PlanningGraph:
75         def __init__(self, problem, state, serialize=True, ignore_preconditions=False):
76             """
77             Parameters
78             -----
79             problem : PlanningProblem
80                 An instance of the PlanningProblem class
81
82             state : tuple(bool)
83                 An ordered sequence of True/False values indicating the truth value
84                 of the corresponding fluent in problem.state_map
85
86             serialize : bool
87                 Flag indicating whether to serialize non-persistent literals. If False,
88                 should NOT be serialized for regression search (e.g. for heuristic
89                 _should_be serialized if the planning graph is built using
90                 a heuristic
91             """

```

```

91
92     self._serialize = serialize
93     self._is_leveled = False
94     self._ignore_mutexes = ignore_mutexes
95     self.goal = set(problem.goal)
96
97     # make no-op actions that persist every literal to the
98     no_ops = [make_node(n, no_op=True) for n in chain(*(problem.literals))]
99     self._actionNodes = no_ops + [make_node(a) for a in problem.actions]
100
101     # initialize the planning graph by finding the literals in the
102     # first layer and finding the actions they they should be applied to
103     literals = [s if f else ~s for f, s in zip(state, problem.literals)]
104     layer = LiteralLayer(literals, ActionLayer(), self._ignore_mutexes)
105     layer.update_mutexes()
106     self.literal_layers = [layer]
107     self.action_layers = []
108
109     def h_levelsum(self):
110         """ Calculate the level sum heuristic for the planning problem.
111
112         The level sum is the sum of the level costs of all the goal literals
113         combined. The "level cost" to achieve any single goal literal is the
114         level at which the literal first appears in the planning graph. Note that
115         the level cost is **NOT** the minimum number of actions required to
116         achieve a single goal literal.
117
118         For example, if Goal_1 first appears in level 0 of the planning graph
119         (it is satisfied at the root of the planning graph) and Goal_2 first
120         appears in level 3, then the levelsum is 0 + 3 = 3.
121
122         Hints
123         -----
124         (1) See the pseudocode folder for help on a simple implementation.
125         (2) You can implement this function more efficiently by using the
126             sample pseudocode if you expand the graph one layer at a time
127             and accumulate the level cost of each goal rather than building
128             the whole graph at the start.
129
130         See Also
131         -----
132         Russell-Norvig 10.3.1 (3rd Edition)
133         """
134         self.fill()
135
136         i = 0
137         for g in self.goal:
138             for idx, layer in enumerate(self.literal_layers):
139                 if g in layer:
140                     i += idx
141                     break
142         return i

```

AWESOME

Excellent

```

144     def h_maxlevel(self):
145         """ Calculate the max level heuristic for the planning
146
147         The max level is the largest level cost of any single
148         The "level cost" to achieve any single goal literal is
149         which the literal first appears in the planning graph
150         the level cost is **NOT** the minimum number of actions
151         a single goal literal.
152
153         For example, if Goal1 first appears in level 1 of the
154         Goal2 first appears in level 3, then the levelsum is
155
156         Hints
157         -----
158         (1) See the pseudocode folder for help on a simple
159         (2) You can implement this function more efficiently
160             the graph one level at a time until the last goal
161             than filling the whole graph at the start.
162
163         See Also
164         -----
165         Russell-Norvig 10.3.1 (3rd Edition)
166
167         Notes
168         -----
169         WARNING: you should expect long runtimes using this heuristic
170         """
171         # maxlevel heuristic
172         self.fill()
173         i = 0
174         for g in self.goal:
175             for idx, layer in enumerate(self.literal_layers):
176                 if g in layer:
177                     i = max(i, idx)
178                     break
179         return i
180
181     def h_setlevel(self):
182         """ Calculate the set level heuristic for the planning
183
184         The set level of a planning graph is the first level
185         appear such that no pair of goal literals are mutex in
186         layer of the planning graph.
187
188         Hints
189         -----
190         (1) See the pseudocode folder for help on a simple
191         (2) You can implement this function more efficiently
192             the graph one level at a time until you find the
193             than filling the whole graph at the start.
194
195         See Also
196         -----
197         Russell-Norvig 10.3.1 (3rd Edition)
198
199         Notes
200         -----
201         WARNING: you should expect long runtimes using this heuristic
202         """
203         # setlevel heuristic

```

```

203
204     def all_goals(layer):
205         for g in self.goal:
206             if g not in layer:
207                 return False
208         return True
209
210     def no_mutex(layer):
211         for g1, g2 in combinations(self.goal, 2):
212             if layer.is_mutex(g1, g2):
213                 return False
214         return True
215
216     level = 0
217     while not self._is_leveled:
218         layer = self.literal_layers[-1]
219         if all_goals(layer) and no_mutex(layer):
220             return level
221
222         self._extend()
223         level += 1

```

AWESOME

Brilliant.

```

224         return -1
225
226
227     #####
228     # DO NOT MODIFY CODE BELOW THIS LINE
229     #####
230
231     def fill(self, maxlevels=-1):
232         """ Extend the planning graph until it is leveled, or
233         levels have been added
234
235         Parameters
236         -----
237         maxlevels : int
238             The maximum number of levels to extend before breaking
239             a negative value will never interrupt the loop.)
240
241         Notes
242         ----
243         YOU SHOULD NOT THIS FUNCTION TO COMPLETE THE PROJECT,
244         """
245         while not self._is_leveled:
246             if maxlevels == 0: break
247             self._extend()
248             maxlevels -= 1
249         return self
250
251     def _extend(self):
252         """ Extend the planning graph by adding both a new action
253
254         The new action layer contains all actions that could
255         negative literals in the leaf nodes of the parent literal

```

```

256
257     The new literal layer contains all literals that could
258     action in the NEW action layer.
259     """
260     if self._is_levelled: return
261
262     parent_literals = self.literal_layers[-1]
263     parent_actions = parent_literals.parent_layer
264     action_layer = ActionLayer(parent_actions, parent_literals)
265     literal_layer = LiteralLayer(parent_literals, action_layer)
266
267     for action in self._actionNodes:
268         # actions in the parent layer are skipped because
269         # which is performed automatically in the ActionLayer
270         if action not in parent_actions and action.preconditions:
271             action_layer.add(action)
272             literal_layer |= action.effects
273
274         # add two-way edges in the graph connecting the
275         parent_literals.add_outbound_edges(action, action_layer)
276         action_layer.add_inbound_edges(action, action_layer)
277
278         # # add two-way edges in the graph connecting the
279         action_layer.add_outbound_edges(action, action_layer)
280         literal_layer.add_inbound_edges(action, action_layer)
281
282     action_layer.update_mutexes()
283     literal_layer.update_mutexes()
284     self.action_layers.append(action_layer)
285     self.literal_layers.append(literal_layer)
286     self._is_levelled = literal_layer == action_layer.parent_layer
287

```

AWESOME

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

RETURN TO PATH