



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

```

1 from aimacode.logic import PropKB
2 from aimacode.planning import Action
3 from aimacode.search import (
4     Node, Problem,
5 )
6 from aimacode.utils import expr
7 from lp_utils import (
8     FluentState, encode_state, decode_state,
9 )
10 from my_planning_graph import PlanningGraph
11
12 from functools import lru_cache
13
14
15 class AirCargoProblem(Problem):
16     def __init__(self, cargos, planes, airports, initial: FluentState, goal: list):
17         """
18         :param cargos: list of str
19             cargos in the problem
20         :param planes: list of str
21             planes in the problem
22         :param airports: list of str
23             airports in the problem
24         :param initial: FluentState object
25             positive and negative literal fluents (as expr) describing initial state
26         :param goal: list of expr
27             literal fluents required for goal test
28         """
29         self.state_map = initial.pos + initial.neg
30         self.initial_state_TF = encode_state(initial, self.state_map)
31         Problem.__init__(self, self.initial_state_TF, goal=goal)
32         self.cargos = cargos
33         self.planes = planes
34         self.airports = airports
35         self.actions_list = self.get_actions()
36
37     def get_actions(self):
38         """
39         This method creates concrete actions (no variables) for all actions in the problem
40         domain action schema and turns them into complete Action objects as defined in the
41         aimacode.planning module. It is computationally expensive to call this method directly;
42         however, it is called in the constructor and the results cached in the `actions_list` property.
43
44         Returns:
45         -----
46         list<Action>
47             list of Action objects
48         """
49
50         # TODO create concrete Action objects based on the domain action schema for: Load, Unload, and Fly
51         # concrete actions definition: specific literal action that does not include variables as with the schema
52         # for example, the action schema 'Load(c, p, a)' can represent the concrete actions 'Load(C1, P1, SFO)'
53         # or 'Load(C2, P2, JFK)'. The actions for the planning problem must be concrete because the problems in
54         # forward search and Planning Graphs must use Propositional Logic
55
56     def load_actions():
57         """Create all concrete Load actions and return a list
58
59         :return: list of Action objects
60         """
61         loads = []
62         # TODO create all load ground actions from the domain Load action
63         for c in self.cargos:
64             for p in self.planes:

```

```

65         for a in self.airports:
66             precondition_pos = [expr("At({}, {})".format(c, a)), expr("At({}, {})".format(p, a))]
67             precondition_neg = []
68             effect_add = [expr("In({}, {})".format(c, p))]
69             effect_rem = [expr("At({}, {})".format(c, a))]
70             load = Action(expr("Load({}, {}, {})".format(c, p, a)),
71                           [precondition_pos, precondition_neg],
72                           [effect_add, effect_rem])
73             loads.append(load)
74         return loads
75
76 def unload_actions():
77     """Create all concrete Unload actions and return a list
78
79     :return: list of Action objects
80     """
81     unloads = []
82     # TODO create all Unload ground actions from the domain Unload action
83     for c in self.cargos:
84         for p in self.planes:
85             for a in self.airports:
86                 precondition_pos = [expr("In({}, {})".format(c, p)), expr("At({}, {})".format(p, a))]
87                 precondition_neg = []
88                 effect_add = [expr("At({}, {})".format(c, a))]
89                 effect_rem = [expr("In({}, {})".format(c, p))]
90                 unload = Action(expr("Unload({}, {}, {})".format(c, p, a)),
91                                [precondition_pos, precondition_neg],
92                                [effect_add, effect_rem])
93                 unloads.append(unload)
94     return unloads
95
96 def fly_actions():
97     """Create all concrete Fly actions and return a list
98
99     :return: list of Action objects
100    """
101    flies = []
102    for fr in self.airports:
103        for to in self.airports:
104            if fr != to:
105                for p in self.planes:
106                    precondition_pos = [expr("At({}, {})".format(p, fr)),
107                                        ]
108                    precondition_neg = []
109                    effect_add = [expr("At({}, {})".format(p, to))]
110                    effect_rem = [expr("At({}, {})".format(p, fr))]
111                    fly = Action(expr("Fly({}, {}, {})".format(p, fr, to)),
112                                [precondition_pos, precondition_neg],
113                                [effect_add, effect_rem])
114                    flies.append(fly)
115    return flies
116
117    return load_actions() + unload_actions() + fly_actions()
118
119 def actions(self, state: str) -> list:
120     """ Return the actions that can be executed in the given state.
121
122     :param state: str
123         state represented as T/F string of mapped fluents (state variables)
124         e.g. 'FTTTF'
125     :return: list of Action objects
126     """
127     # TODO implement
128     # create knowledge base
129     kb = PropKB()
130     # decode string of T/F as fluent per mapping
131     FluentState = decode_state(state, self.state_map)
132     # Add clauses of FluentState's positive sentence to the KB
133     kb.tell(FluentState.pos_sentence())
134     '''
135     #one version to complete this function is to return the variable in the folloing line
136     possible_actions = [ a for a in self.actions_list if a.check_precond(kb, a.args) ]
137     '''
138     #however, we use this longer version that run faster in our experiments
139     #probably because it didn't call the action method check_precond from aimacode.planning
140     possible_actions = []
141     for action in self.actions_list:
142         executable = True
143         for clause in action.precond_pos:
144             if clause not in kb.clauses:
145                 executable = False
146         for clause in action.precond_neg:
147             if clause in kb.clauses:
148                 executable = False
149         if executable:
150             possible_actions.append(action)
151     return possible_actions
152
153 def result(self, state: str, action: Action):
154     """ Return the state that results from executing the given
155     action in the given state. The action must be one of
156     self.actions(state).
157
158     :param state: state entering node
159     :param action: Action applied
160     :return: resulting state after action
161     """

```

```

163     # TODO implement
164     new_state = FluentState([], [])
165     old_state = decode_state(state, self.state_map)
166     #add-list: 1. unaffected positive literals from old state
167     for fluent in old_state.pos:
168         if fluent not in action.effect_rem:
169             new_state.pos.append(fluent)
170     #add-list: 2. positive literals in the action's effect
171     for fluent in action.effect_add:
172         if fluent not in new_state.pos:
173             new_state.pos.append(fluent)
174     #delete-list: 1. unaffected negative literals from old state
175     for fluent in old_state.neg:
176         if fluent not in action.effect_add:
177             new_state.neg.append(fluent)
178     #delete-list: 2. negative literals in the action's effect
179     for fluent in action.effect_rem:
180         if fluent not in new_state.neg:
181             new_state.neg.append(fluent)
182     return encode_state(new_state, self.state_map)
183
184
185 def goal_test(self, state: str) -> bool:
186     """ Test the state to see if goal is reached
187
188     :param state: str representing state
189     :return: bool
190     """
191     kb = PropKB()
192     kb.tell(decode_state(state, self.state_map).pos_sentence())
193     for clause in self.goal:
194         if clause not in kb.clauses:
195             return False
196     return True
197
198 def h_1(self, node: Node):
199     # note that this is not a true heuristic
200     h_const = 1
201     return h_const
202
203 @lru_cache(maxsize=8192)
204 def h_pg_levelsum(self, node: Node):
205     """This heuristic uses a planning graph representation of the problem
206     state space to estimate the sum of all actions that must be carried
207     out from the current state in order to satisfy each individual goal
208     condition.
209     """
210     # requires implemented PlanningGraph class
211     pg = PlanningGraph(self, node.state)
212     pg_levelsum = pg.h_levelsum()
213     return pg_levelsum
214
215 @lru_cache(maxsize=8192)
216 def h_ignore_preconditions(self, node: Node):
217     """This heuristic estimates the minimum number of actions that must be
218     carried out from the current state in order to satisfy all of the goal
219     conditions by ignoring the preconditions required for an action to be
220     executed.
221     """
222     # TODO implement (see Russell-Norvig Ed-3 10.2.3 or Russell-Norvig Ed-2 11.2)
223     count = 0
224     kb = PropKB()
225     #decode string of T/F as fluent per mapping
226     FluentState = decode_state(node.state, self.state_map)
227     #Add clauses of FluentState's positive sentence to the KB
228     kb.tell(FluentState.pos_sentence())
229     #for clause in kb.clauses:
230
231     for clause in self.goal:
232         if clause not in kb.clauses:
233             #sometimes it require more than one action to reach the goal, but under-estimating is admissible
234             #by this (inaccurate) counting, this heuristic is sometimes called number-of-unsatisfied-goals heuristic
235             count += 1
236     return count
237
238
239 def air_cargo_pl() -> AirCargoProblem:
240     cargos = ['C1', 'C2']
241     planes = ['P1', 'P2']
242     airports = ['JFK', 'SFO']
243     pos = [expr('At(C1, SFO)'),
244            expr('At(C2, JFK)'),
245            expr('At(P1, SFO)'),
246            expr('At(P2, JFK)'),
247            ]
248     neg = [expr('At(C2, SFO)'),
249            expr('In(C2, P1)'),
250            expr('In(C2, P2)'),
251            expr('At(C1, JFK)'),
252            expr('In(C1, P1)'),
253            expr('In(C1, P2)'),
254            expr('At(P1, JFK)'),
255            expr('At(P2, SFO)'),
256            ]
257     init = FluentState(pos, neg)
258     goal = [expr('At(C1, JFK)'),
259             expr('At(C2, SFO)'),
260             ]
261     return AirCargoProblem(cargos, planes, airports, init, goal)

```

```

262
263
264 def air_cargo_p2() -> AirCargoProblem:
265     # TODO implement Problem 2 definition
266     cargos = ['C1', 'C2', 'C3']
267     planes = ['P1', 'P2', 'P3']
268     airports = ['JFK', 'SFO', 'ATL']
269     #positive fluents
270     pos = [expr('At(C1, SFO)'),
271            expr('At(C2, JFK)'),
272            expr('At(C3, ATL)'),
273            expr('At(P1, SFO)'),
274            expr('At(P2, JFK)'),
275            expr('At(P3, ATL)'),
276            ]
277     #negative fluents
278     neg = []
279     for c in cargos:
280         for a in airports:
281             expression = expr('At({}, {})'.format(c, a))
282             if expression not in pos:
283                 neg.append(expression)
284     for p in planes:
285         for a in airports:
286             expression = expr('At({}, {})'.format(p, a))
287             if expression not in pos:
288                 neg.append(expression)
289     #none of the cargo had been loaded
290     for c in cargos:
291         for p in planes:
292             neg.append(expr('In({}, {})'.format(c, p)))
293     init = FluentState(pos, neg)
294     goal = [expr('At(C1, JFK)'),
295            expr('At(C2, SFO)'),
296            expr('At(C3, SFO)'),
297            ]
298     print('init=', init)
299     print('init.sentence=', init.sentence())
300     print('init.pos_sentence=', init.pos_sentence())
301     print('init.pos=', init.pos)
302     print('goal=', goal)
303
304     return AirCargoProblem(cargos, planes, airports, init, goal)
305
306
307 def air_cargo_p3() -> AirCargoProblem:
308     # TODO implement Problem 3 definition
309     cargos = ['C1', 'C2', 'C3', 'C4']
310     planes = ['P1', 'P2']
311     airports = ['JFK', 'SFO', 'ATL', 'ORD']
312     #positive fluents
313     pos = [expr('At(C1, SFO)'),
314            expr('At(C2, JFK)'),
315            expr('At(C3, ATL)'),
316            expr('At(C4, ORD)'),
317            expr('At(P1, SFO)'),
318            expr('At(P2, JFK)'),
319            ]
320     #negative fluents
321     neg = []
322     for c in cargos:
323         for a in airports:
324             expression = expr('At({}, {})'.format(c, a))
325             if expression not in pos:
326                 neg.append(expression)
327     for p in planes:
328         for a in airports:
329             expression = expr('At({}, {})'.format(p, a))
330             if expression not in pos:
331                 neg.append(expression)
332     #none of the cargo had been loaded
333     for c in cargos:
334         for p in planes:
335             neg.append(expr('In({}, {})'.format(c, p)))
336     init = FluentState(pos, neg)
337     goal = [expr('At(C1, JFK)'),
338            expr('At(C2, SFO)'),
339            expr('At(C3, JFK)'),
340            expr('At(C4, SFO)'),
341            ]
342     return AirCargoProblem(cargos, planes, airports, init, goal)
343
344

```

AWESOME

Good work on implementing all the problem definitions.

RETURN TO PATH

Rate this review

[Student FAQ](#)