

1 Classical Planning Overview

1.1 Background

Planning, in general, consists of computing a sequence of actions which, starting in an initial state or states, will achieve a set of goal states. Classical planning, then, is a constrained planning task, which is fully observable, deterministic, finite, static (i.e. the world does not change unless the agent acts), and discrete (i.e. time, actions, objects and effects are not continuous). Nonclassical planning, by contrast, is used to describe planning in partially observable or stochastic environments.

1.1.1 Difficulties of Classical Planning

Typical problem-solving agents which make use of standard search algorithms (i.e. depth-first, breadth-first, A^* , etc) encounter several difficulties when solving classical planning problems; each of these must be kept in mind when designing a planning agent. First of all, the agent may be overwhelmed by “irrelevant” actions (that is, actions which do not necessarily bring the agent closer to achieving a goal). There are multiple ways within the classical planning community of discouraging a planning agent from considering irrelevant actions; the use of regression search, for example, is one way to attempt to handle this problem.

Another difficulty in classical planning is finding a good heuristic function. Because a problem-solving agent would see the goal state only as a “black box” which is either true or false for a given state, it would require a new heuristic for each specific problem it encounters. A planning agent, however, may have access to a representation of the goal state as a conjunction of subgoals, in which case it can use a single domain-independent heuristic; namely, the number of unsatisfied conjuncts.

Similarly, a problem-solving agent might be inefficient because of its inability to take advantage of problem decomposition. A planning agent which views a state as a conjunction of literals, however, might be able to decompose the goal into a conjunction of subgoals, and then attempt to find a plan to achieve each subgoal. There are tradeoffs to this approach, however. Even if a planning agent can work on subgoals independently, some additional work may be required to combine the resultant subplans. Additionally, for some domains, on one subgoal may undo another.

1.1.2 Representing Classical Planning Problems

Classical planning problems are represented using three formalisms: states, goals and actions. These constructs form the basis of the languages used to encode planning problems in such a way that planners can understand them and operate on them to search for plans.

States in classical planning problems are represented by a conjunction of positive, ground, first-order propositional literals. Classical planning adopts the closed-world assumption, which states that any literals which are not explicitly enumerated as part of the state are false. This state representation may be manipulated by logical inference (when viewed as a conjunction of fluents), or with set operations (when viewed as a set of fluents).

A goal in a classical planning problem is a partially specified state, represented by a conjunction of ground literals. For some propositional state s , we say that s satisfies a goal g if all of the atoms in g are in s (where s may also contain other atoms).

Actions are specified with respect to their preconditions and effects. The precondition of an action is a conjunction of positive literals which must be true in a state before the action can be executed. The effect of an action is a conjunction of literals which occur as a result of executing the action. For some literal P in the effect, P is asserted to be true in the resulting state; likewise, if $\neg P$ is in the effect, P is asserted to be false in the resulting state. It is worth noting that this action representation avoids the frame problem by leaving every literal not in an action's effect unchanged as a result of executing that action.

1.2 PDDL

1.3 FF Planner

1.4 Planning Domain Examples

1.4.1 Blocks World

Listing 1: Blocks World Domain Description in PDDL

```

1 (define (domain BLOCKS)
2   (:requirements :strips :typing)
3   (:types block)
4   (:predicates (on ?x - block ?y - block)
5     (ontable ?x - block)
6     (clear ?x - block)
7     (handempty)
8     (holding ?x - block)
9   )
10
11   (:action pick-up
12     :parameters (?x - block)
13     :precondition (and (clear ?x) (ontable ?x) (handempty))
14     :effect (and
15       (not (ontable ?x))
16       (not (clear ?x))
17       (not (handempty))
18       (holding ?x)
19     )
20   )
21
22   (:action put-down
23     :parameters (?x - block)
24     :precondition (holding ?x)
25     :effect (and
26       (not (holding ?x))
27       (clear ?x)
28       (handempty)
29       (ontable ?x)
30     )
31   )
32
33   (:action stack
34     :parameters (?x - block ?y - block)
35     :precondition (and (holding ?x) (clear ?y))
36     :effect (and
37       (not (holding ?x))
38       (not (clear ?y))
39       (clear ?x)

```

```

40         (handempty)
41         (on ?x ?y)
42     )
43 )
44
45 (:action unstack
46   :parameters (?x - block ?y - block)
47   :precondition (and (on ?x ?y) (clear ?x) (handempty))
48   :effect (and
49     (holding ?x)
50     (clear ?y)
51     (not (clear ?x))
52     (not (handempty))
53     (not (on ?x ?y))
54   )
55 )
56 )

```

1.4.2 Towers of Hanoi

Listing 2: Towers of Hanoi Domain Description in PDDL

```

1  ;; towers of hanoi
2
3  (define (domain HANOI)
4    (:requirements :typing)
5    (:types disc peg)
6    (:predicates
7      (clear ?x)
8      (on ?x - disc ?y)
9      (larger ?d - disc ?e - disc)
10   )
11
12   (:action stack-d
13     :parameters (?d - disc ?e - disc)
14     :vars (?l)
15     :precondition (and
16       (on ?d ?l)
17       (not (on ?d ?e))
18       (not (= ?d ?e))
19       (not (= ?e ?l))
20       (larger ?e ?d)
21       (clear ?d)
22       (clear ?e)
23     )
24     :effect (and
25       (not (on ?d ?l))
26       (not (clear ?e))
27       (on ?d ?e)
28       (clear ?l)
29     )
30   )
31
32   (:action stack-p
33     :parameters (?d - disc ?p - peg)
34     :vars (?l)
35     :precondition (and
36       (on ?d ?l)
37       (clear ?p)
38       (clear ?d)

```

```

39         (not (= ?p ?l))
40     )
41     :effect (and
42         (not (clear ?p))
43         (not (on ?d ?l))
44         (on ?d ?p)
45         (clear ?l)
46     )
47 )
48 )

```

1.4.3 Lin's Briefcase

Listing 3: Lin's Briefcase Domain Description in PDDL

```

1  ;; briefcase domain
2
3  (define (domain BRIEFCASE)
4      (:requirements :typing)
5      (:types latch)
6      (:predicates
7          (open)
8          (latched ?l - latch)
9      )
10
11     (:action flip-open
12         :parameters (?l - latch)
13         :precondition (latched ?l)
14         :effect (not (latched ?l))
15     )
16
17     (:action flip-closed
18         :parameters (?l - latch)
19         :precondition (not (latched ?l))
20         :effect (latched ?l)
21     )
22
23     (:action open
24         :parameters ()
25         :precondition (and
26             (forall (?l - latch)
27                 (not (latched ?l))
28             )
29             (not (open))
30         )
31         :effect (open)
32     )
33 )

```

1.4.4 Electrical Circuit

Listing 4: Electrical Circuit Domain Description in PDDL

```

1  ;; circuit domain
2
3  (define (domain CIRCUIT)
4      (:requirements :typing :conditional-effects)
5      (:types wire gate level)
6      (:predicates
7          (wire-high ?w - wire)

```

```

8      (gate-active ?g - gate)
9      (gate-level ?g - gate ?l - level)
10     (and-gate ?g - gate)
11     (or-gate ?g - gate)
12     (inv-gate ?g - gate)
13     (input-to ?w - wire ?g - gate)
14     (output-from ?w - wire ?g - gate)
15 )
16
17 (:action activate-wire
18   :parameters (?w - wire)
19   :vars (?g2 - gate)
20   :precondition (and
21     (not (wire-high ?w))
22     (input-to ?w ?g2)
23     (forall (?g - gate)
24       (not (output-from ?w ?g))
25     )
26   )
27   :effect (and
28     (wire-high ?w)
29   )
30 )
31
32 (:action deactivate-wire
33   :parameters (?w - wire)
34   :vars (?g2 - gate)
35   :precondition (and
36     (wire-high ?w)
37     (input-to ?w ?g2)
38     (forall (?g - gate)
39       (not (output-from ?w ?g))
40     )
41   )
42   :effect (and
43     (not (wire-high ?w))
44   )
45 )
46 (:action activate-and-gate
47   :parameters (?g - gate)
48   :vars (?w1 - wire ?w2 - wire ?w3 - wire)
49   :precondition (and
50     (and-gate ?g)
51     (not (gate-active ?g))
52     (input-to ?w1 ?g)
53     (input-to ?w2 ?g)
54     (output-from ?w3 ?g)
55     (wire-high ?w1)
56     (wire-high ?w2)
57     (not (= ?w1 ?w2))
58     (not (= ?w1 ?w3))
59     (not (= ?w2 ?w3))
60   )
61   :effect (and
62     (wire-high ?w3)
63     (gate-active ?g)
64   )
65 )
66
67 (:action activate-inv-gate
68   :parameters (?g - gate)

```

```

69      :vars (?w1 - wire ?w2 - wire)
70      :precondition (and
71        (inv-gate ?g)
72        (not (gate-active ?g))
73        (input-to ?w1 ?g)
74        (output-from ?w2 ?g)
75        (not (wire-high ?w1))
76      )
77      :effect (and
78        (wire-high ?w2)
79        (gate-active ?g)
80      )
81    )
82
83    (:action activate-or-gate
84      :parameters (?g - gate)
85      :vars (?w1 - wire ?w2 - wire ?w3 - wire)
86      :precondition (and
87        (or-gate ?g)
88        (not (gate-active ?g))
89        (input-to ?w1 ?g)
90        (input-to ?w2 ?g)
91        (output-from ?w3 ?g)
92        (or (wire-high ?w1) (wire-high ?w2))
93      )
94      :effect (and
95        (wire-high ?w3)
96        (gate-active ?g)
97      )
98    )
99  )

```