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).

Actions may be represented by an action schema. This schema consists of the action name, a list of all variables used in the schema, a precondition, and an effect. Both the precondition and effect of the action are conjunctions of literals, which may be either positive or negative. The precondition gives a definition of states in which the action can be executed; the effect gives the result of executing the action. More formally, an action a may be executed in some state s if s entails the precondition of a. If an action schema contains variables, there may be multiple states (ground instantiations of the variables in the action) which satisfy the preconditions of the action.

The result of executing some action a in state s is given by state s'. s' is first populated with the set of fluents in s, then all fluents which appear as negative literals in the effects of a are removed, and finally the fluents which appear as positive literals in the effects of a are added.

A set of these action schemas gives the definition of a planning domain. A specific classical planning problem within such a domain is defined by adding an initial state and a goal. The initial state is a conjunction of ground atoms (where the closed-world assumption is assumed to hold). The goal is defined much like the precondition of an action; it is a conjunction of literals (either positive or negative) which may contain (existentially qualified) variables. A solution to a classical planning problem is a sequence of actions which begin with the initial state, and which entails the goal.

1.2 PDDL

The Planning Domain Definition Language, or PDDL, is the lingua franca in the classical planning community. It was originally developed by the Artificial Intelligence Planning Systems 1998 Competition Committee for use in defining problem domains. It is not as expressive as some other languages which may be used for planning; however, this is by design, as the planning community wanted the language to be as simple and efficient as possible. Because it was designed for use in an international planning competition, development of the syntax and semantics of PDDL has been largely tied to various competitions as well. As a result, the development of planners which are able to execute PDDL code to search for plans has been somewhat piecemeal with respect to the formal language specification. That is, particular planners submitted to these competitions may only implement a subset of the features given in the language specification of PDDL.

1.3 FF Planner

In order to test the PDDL code we wrote for each of our test domains, we needed to decide on a planner to use. We selected Fast-Forward (FF), because it was simple to compile and run on our systems, and it supported most of the features of PDDL we wanted to be able to take advantage of when encoding the test domains.

FF was the most successful automatic planner in the AIPS-2000 planning systems competition. However, the general idea behind FF was not new to the classical planning community - the basic principle is actually the same as that of the Heuristic Search Planner (HSP). FF executes a forward search in the state space, guided by a heuristic which is extracted from the domain description automatically. This function is extracted by relaxing the planning problem; a part of the specification (specifically, the delete lists of all actions) is ignored.

There are a number of details in which FF is different from its predecessor, HSP:

1. FF makes use of a more sophisticated method of heuristic evaluation, which takes into account positive interactions between facts.
2. FF uses a different local search strategy; specifically, it is able to escape plateaus and local minima through the use of systematic search.

3. FF includes a mechanism which identifies those successors of a search node which appear to be (and usually are) most helpful in achieving the goal.

The main difficulty which results from viewing domain independent planning as heuristic search is the automated derivation of the heuristic function. A common approach to this problem (which is adopted here) is to relax the general problem $\mathcal{P}$ into a simpler problem $\mathcal{P}'$ which can be efficiently solved. Given a search state in the original problem, $\mathcal{P}$, the solution length of the same state $\mathcal{P}'$ may be used to estimate the difficulty of $\mathcal{P}$.

1.4 Planning Domain Examples

1.4.1 Blocks World

Listing 1: Blocks World Domain Description in PDDL

```
(define (domain BLOCKS)
    (:requirements :strips :typing)
    (:types block)
    (:predicates (on ?x - block ?y - block)
        (ontable ?x - block)
        (clear ?x - block)
        (handempty)
        (holding ?x - block)
    )

    (:action pick-up
        :parameters (?x - block)
        :precondition (and (clear ?x) (ontable ?x) (handempty))
        :effect (and
            (not (ontable ?x))
            (not (clear ?x))
            (not (handempty))
            (holding ?x)
        )
    )

    (:action put-down
        :parameters (?x - block)
        :precondition (holding ?x)
        :effect (and
            (not (holding ?x))
            (clear ?x)
            (handempty)
            (ontable ?x)
        )
    )

    (:action stack
        :parameters (?x - block ?y - block)
        :precondition (and (holding ?x) (clear ?y))
        :effect (and
            (not (holding ?x))
            (not (clear ?y))
            (clear ?x)
            (handempty)
            (on ?x ?y)
        )
    )
```

```
    (:action unstack
        :parameters (?x - block ?y - block)
        :precondition (and (on ?x ?y) (clear ?x) (handempty))
        :effect (and
            (holding ?x)
            (clear ?y)
            (not (clear ?x))
            (not (handempty))
            (not (on ?x ?y))
        )
    )
)
```

1.4.2 Towers of Hanoi

Listing 2: Towers of Hanoi Domain Description in PDDL

```
;; towers of hanoi

(define (domain HANOI)
    (:requirements :typing)
    (:types disc peg)
    (:predicates
        (clear ?x)
        (on ?x - disc ?y)
        (larger ?d - disc ?e - disc)
    )

    (:action stack-d
        :parameters (?d - disc ?e - disc)
        :vars (?l)
        :precondition (and
            (on ?d ?l)
            (not (on ?d ?e))
            (not (= ?d ?e))
            (not (= ?e ?l))
            (larger ?e ?d)
            (clear ?d)
            (clear ?e)
        )
        :effect (and
            (not (on ?d ?l))
            (not (clear ?e))
            (on ?d ?e)
            (clear ?l)
        )
    )

    (:action stack-p
        :parameters (?d - disc ?p - peg)
        :vars (?l)
        :precondition (and
            (on ?d ?l)
            (clear ?p)
            (clear ?d)
            (not (= ?p ?l))
        )
        :effect (and
            (not (clear ?p))
            (not (on ?d ?l))
```

```
            (on ?d ?p)
            (clear ?l)
        )
    )
)
```

1.4.3 Lin's Briefcase

Listing 3: Lin's Briefcase Domain Description in PDDL

```
;; briefcase domain

(define (domain BRIEFCASE)
    (:requirements :typing)
    (:types latch)
    (:predicates
        (open)
        (latched ?l - latch)
    )

    (:action flip-open
        :parameters (?l - latch)
        :precondition (latched ?l)
        :effect (not (latched ?l))
    )

    (:action flip-closed
        :parameters (?l - latch)
        :precondition (not (latched ?l))
        :effect (latched ?l)
    )

    (:action open
        :parameters ()
        :precondition (and
            (forall (?l - latch)
                (not (latched ?l))
            )
            (not (open))
        )
        :effect (open)
    )
)
```

1.4.4 Electrical Circuit

Listing 4: Electrical Circuit Domain Description in PDDL

```
;; circuit domain

(define (domain CIRCUIT)
    (:requirements :typing :conditional-effects)
    (:types wire gate level)
    (:predicates
        (wire-high ?w - wire)
        (gate-active ?g - gate)
        (gate-level ?g - gate ?l - level)
        (and-gate ?g - gate)
        (or-gate ?g - gate)
        (inv-gate ?g - gate)
```

```
        (input-to ?w - wire ?g - gate)
        (output-from ?w - wire ?g - gate)
    )

    (:action activate-wire
        :parameters (?w - wire)
        :vars (?g2 - gate)
        :precondition (and
            (not (wire-high ?w))
            (input-to ?w ?g2)
            (forall (?g - gate)
                (not (output-from ?w ?g))
            )
        )
        :effect (and
            (wire-high ?w)
        )
    )

    (:action deactivate-wire
        :parameters (?w - wire)
        :vars (?g2 - gate)
        :precondition (and
            (wire-high ?w)
            (input-to ?w ?g2)
            (forall (?g - gate)
                (not (output-from ?w ?g))
            )
        )
        :effect (and
            (not (wire-high ?w))
        )
    )
    (:action activate-and-gate
        :parameters (?g - gate)
        :vars (?w1 - wire ?w2 - wire ?w3 - wire)
        :precondition (and
            (and-gate ?g)
            (not (gate-active ?g))
            (input-to ?w1 ?g)
            (input-to ?w2 ?g)
            (output-from ?w3 ?g)
            (wire-high ?w1)
            (wire-high ?w2)
            (not (= ?w1 ?w2))
            (not (= ?w1 ?w3))
            (not (= ?w2 ?w3))
        )
        :effect (and
            (wire-high ?w3)
            (gate-active ?g)
        )
    )

    (:action activate-inv-gate
        :parameters (?g - gate)
        :vars (?w1 - wire ?w2 - wire)
        :precondition (and
            (inv-gate ?g)
            (not (gate-active ?g))
            (input-to ?w1 ?g)
```

```
            (output-from ?w2 ?g)
            (not (wire-high ?w1))
        )
        :effect (and
            (wire-high ?w2)
            (gate-active ?g)
        )
    )

    (:action activate-or-gate
        :parameters (?g - gate)
        :vars (?w1 - wire ?w2 - wire ?w3 - wire)
        :precondition (and
            (or-gate ?g)
            (not (gate-active ?g))
            (input-to ?w1 ?g)
            (input-to ?w2 ?g)
            (output-from ?w3 ?g)
            (or (wire-high ?w1) (wire-high ?w2))
        )
        :effect (and
            (wire-high ?w3)
            (gate-active ?g)
        )
    )
)
```