The 1998 AI Planning Systems Competition

Drew McDermott

April 17, 2000

Abstract

The 1998 Planning Competition at the AI Planning Systems Conference was the first of its kind. Its goal was to create planning domains that a wide variety of planning researchers could agree on, so as to make comparison among planners more meaningful, measure overall progress in the field, and set up a framework for long-term creation of a repository of problems in a standard notation. A rules committee for the competition was created in 1997, and had long discussions on how the contest should go. One result of those discussions was the PDDL notation for planning domains. This notation was used to set up a set of planning problems, and to get a modest problem repository started. As a result, five planning systems were able to compete when the contest took place, in June, 1998. All of these systems solved problems in the Strips framework, with some slight extensions. The attempt to find domains for other forms of planning foundered because of technical and organizational problems. In spite of this, the competition achieved its goals partially, in that it confirmed that substantial progress had occurred in some subfields of planning, and it allowed qualitative comparison among different planning algorithms. It is urged that the competition continue to take place and to evolve.

1 History

In recent years, many subfields of AI have used competitions as a way of measuring progress and guiding research directions (e.g., [10, 11, 1]). In a competition, researchers run their programs on a common set of problems at the same time with little tuning, and the results are compared. There are several purposes to such an exercise:

- It allows meaningful comparison of programs.
- It can provide an indication of overall progress in the field.
- It can provide a set of benchmark problems for others to use to compare their systems to the state of the art.
- It can focus attention on more realistic problems.

Of course, competitions have drawbacks. Preparing a program for a competition usually means polishing existing features, and suspending work on new ones. "Realistic problems" may not be those of most interest in the long run. The Message Understanding Competitions focused attention on "information extraction" from newspaper articles or more restricted media. The programs that did well on this task were those that were carefully engineered to detect and dissect messages in the target category. Programs derived from research programs with more ambitious goals, such as investigating the general theory of natural-language understanding, did not work so well. One could argue that this outcome is evidence that such a general theory is, at least for now, a chimera. But many people would disagree, arguing that in the long run the discovery of such a theory is the whole point of studying natural language.

Until the AI Planning Systems (AIPS) conference of 1998, there had never been a competition in the field of automated planning. The broadest definition of *planning* is *reasoning about agent behavior*. A system plans to the extent it predicts the consequences of alternative behaviors before selecting one. Because there are a wide range of agents, reasoning techniques, and ways of combining plan inference with plan execution, the field is quite broad, encompassing everything from factory scheduling to robot programming. Some of the application areas are of immediate practical interest, while others are still rather abstract.

In spite of this immediate practical interest, there haven't been many applications of planners that are actually used. The practical applications tend to lie in three categories:

1. *Scheduling* problems, in which the actions that must be taken are known, and the problem is to find an order in which to carry them out.

2. *Plan management* problems, in which the plans tend to consist of stereotypical structures of actions, usually hierarchical (longer-term actions being decomposed into structures of shorter-term ones), without many choices about how to decompose.

3. *Symbolic control* problems, in which the focus is on execution of plans for controlling a reactive system, where the plans were written by a human.

Applications in these areas tend to succeed because they eliminate or sharply constrain the amount of search the planning system has to do. The focus of the competition was intended to be planning problems that require a significant amount of search. In the past few years, there has been a lot of work on search-based planning algorithms, and a fair amount of progress. It would be premature to say that practical algorithms exist, but planners are now finding solutions to problems that are an order of magnitude larger than those they could solve ten years ago. It seemed as if the time was ripe to hold a competition and see how much progress had really been made, and possibly push the community further in interesting directions.

Although some people (notably Manuela Veloso) had been arguing for a planning competition for years, serious talk about a competition began in 1996, and was a major topic among attendees at the Dagstuhl Workshop on Control of Search in Planning, held in October. The fourth biennial AI Planning Systems (AIPS) Conference was scheduled for June of 1998, and that seemed like the obvious time to have a competition. Veloso was program-co-chair for the conference, and asked me to chair the rules committee for the competition.

By the summer of 1997, we had assembled the following Rules Committee:

- Drew McDermott (chair), Yale University
- Malik Ghallab, Ecole Nationale Superieure D'ingenieur des Constructions Aeronautiques
- Adele Howe, Colorado State University
- Craig Knoblock, University of Southern California
- Ashwin Ram, Georgia Tech
- Manuela Veloso, Carnegie Mellon University
- Daniel Weld, University of Washington
- David Wilkins, SRI

This paper focuses on how the committee designed and ran the competition, with an overall summary of results. This issue of *AI Magazine* contains a companion paper by those who actually competed, with Derek Long serving as overall author and editor. It focuses more on what strengths and weaknesses each planner embodied, with detailed comparison of their performance on various problems.

There were several matters that had to be dealt with in order to have a competition. First, exactly what sort of planning problems would we give people to work on? As suggested above, even after eliminating scheduling and other areas, we are still left with several different types of problem area:

- *Classical planning:* In this kind of problem, you are given an initial situation, a set of action definitions, and a proposition (goal) to be brought about. A solution is a sequence of actions which, when executed beginning in the initial situation, brings about a situation in which the goal is true. It is assumed that the planner knows everything that is true in the initial situation, and knows the effect of every action.
- *Hierararchical planning:* Here you are given, in addition to the material of classical planning, a set of *abstract actions*. An abstract action cannot be executed directly, but must be executed by executing an *expansion* (or *reduction*) of it in terms of less abstract actions, typically one found in a "plan library." A problem may specify, in addition to a goal, an abstract action to be executed. A solution is a sequence of primitive actions that (a) achieves the goal; and (b) corresponds to an expansion of the given abstract action.

- *Reactive planning:* This is a much vaguer classification, in which the assumption of perfect information is relaxed. There are many different types of reactive-planning problem, depending on what is assumed about the sensors and effectors available.

- *Learning in planning:* This is not a problem type so much as an approach to the other types. A learning planner, not surprisingly, does better and better as it gains experience with planning problems of a given type. The style of learning usually studied is *case-based reasoning*, in which new problems are solved by adapting solutions to similar problems previously encountered.

The Rules Committee spent the summer of 1997 in discussion of how to proceed. It seemed clear that it would be hard to find one problem domain that would be attackable by planners in all these categories. Hence it was decided to try to create several "tracks," in which different categories of planner would compete.

Regardless of how many tracks we settled on in the end, it was clear that we would need a notation to use as an input language for the competing programs. So the committee set itself the goal of designing such a language, to be called the Planning Domain Definition Language, or PDDL.

Meanwhile, we engaged in serious discussion of (at least) four "tracks":

1. A Strips track: Classical planning with action-definition notation of the same expressive power as the Strips planner [6].

2. An ADL track: Classical planning with an enhanced notation allowing actions with quantified preconditions and context-dependent effects[12].

3. Hierarchical planning: A classical domain, but with explicitly given compound actions (canned plans) that the planner must be able to reason about ([5, 14]).

4. Reactive planning: Planning in a complex simulated domain, such as the Phoenix fire-fighting simulator [7].

2 PDDL — The Planning Domain Definition Language

The PDDL language was designed to be a neutral specification of planning *problems*. "Neutral" means that it doesn't favor any particular planning system. The slogan we used to summarize this goal, was: "physics, not advice." That is, every piece of a representation would be a necessary part of the specification of what actions were possible and what their effects are. All traces of "hints" to a planning system would be eliminated. For instance, an implication ($P \supset Q$) have several uses, including these:

1. To prove that Q is true in a world situation, adopt proving P as a subgoal.
2. To achieve Q (make it true), adopt achieving P as a subgoal.

A specification of which of these, if either, was actually a good idea would constitute *advice*. On the other hand, consider these two interpretations of $P \supset Q$:

1. In every world situation, either P is false or Q is true.
2. Any plan that causes P to become true without Q being false is invalid no matter what other virtues it has.

The latter interpretation is a *safety* condition, which has a completely different meaning from the former. *This* distinction is one of "physics," although not in the traditional sense. In the first interpretation, $P \supset Q$ is automatically always satisfied. In the second, it might well be violated, but no legal plan is allowed to do so.

We anticipated that most planners would require some kind of advice, because planners search very large spaces and often fail without a bit of help. We didn't want to outlaw advice, just make sure it was properly labeled and accounted for; and omitted entirely from the core language.

It is harder than it sounds to create a notation that avoids advice entirely, for a variety of reasons. Historically, most planning researchers have not made the distinction, so existing input notations are full of it. For example, there are actions that are useful only under certain disagreeable circumstances. If you need three nuts to attach a wheel to an axle, one way to do it is to take one nut from each of the other wheels. This is a reasonable strategy if all other nuts are lost. However, with this action in the database, some planners might consider deliberately throwing all the nuts away in order to make this action feasible. Although they would presumably reject this plan at some point, it might be useful to tell the planner that the condition, "There are no unattached nuts" should be treated only as a "filter" condition, used to select among alternative actions but never to be achieved if false. Many planning researchers have developed sophisticated notations for expressing advice like this, and have made it so easy to include the advice in the action specifications that one is likely to overlook the fact that it's not actually part of the definition of what the action does.

The field in which advice has reigned supreme is in hierarchical planning, which studies planners that assemble solutions from large canned plans stored in some kind of plan library. These plans typically look suspiciously like programs, and some of their steps are essentially procedures for setting up data structures, making sure the plan is appropriate, and so forth. It is hard to extract the pure physics from a representation like that, and just as hard to then represent the advice part as a separate set of hints.

Nonetheless, we felt it was important to try to define a purely physical hierarchical notation, to support the proposed hierarchical-planning track for the competition. Given the way hierarchical planners are often used, this project might be viewed as of questionable sanity. The main reason to have a library of plans in the first place is to focus the planner's attention on those sequences of actions and away from others that are less likely to be useful. In other words, a plan library can be thought of as a hint library. If that's what it is, then specifying it is not part of the PDDL project. A team that wanted to use a plan library would have to treat it as an advice structure superimposed on an underlying physics specified without the use of hierarchy.

However, there is a sense in which a canned plan can be thought of as a purely "physical" entity, and that is when it represents a standard procedure whose functioning can't easily be expressed in more basic terms. A procedure for starting a nuclear reactor is in principle derivable from a more detailed specification of exactly what actions are possible and what their effects are, but that specification may not be available to the planner. It may just know that if it carries out the plan specified in the manual, then it will achieve certain things, and that it can interleave steps of other plans with the steps of this one provided certain constraints are honored.

From this point of view it seemed reasonable to visualize the Hierarchical Planning track of the competition in these terms: for one or more domains we would specify some canned plans, and then stipulate that no problem solution could include any steps that were not part of an instance of one of those plans. In concrete terms, if you had to pump out radioactive waste and restart the reactor, you weren't allowed to flick switch 203 unless one of the protocols for pumping and restarting called for that switch to be flicked at some point, and the other did not forbid it. It turned out that making this precise was extremely difficult, as we will discuss below.

The second most important desideratum in the design of PDDL was that it resemble existing input notations. Most input notations were Lisp-like, for historical reasons, but beyond that there were many divergences. The University of Washington UCPOP input language was the closest thing to a standard. However, it didn't have a simple notation for object types, used Lisp procedures for arithmetic operations, and didn't address the representation of hierarchical plans at all. PDDL was produced by adding to the UCPOP language a simple uniform typing syntax, some arithmetic facilities, and the notion of *action expansion*. Actions were classified as primitive or expandable. An expandable action could not be executed directly, but had to be instantiated by selecting one of its *methods*, each a structure of actions.

Table 1 gives an example of a simple domain, called "Logistics," one of those used in the competition. There were actually two versions, Logistics-adl and Logistics-strips. This one requires the use of ADL constructs like typing and quantification. Types are indicated by hyphens. The type appears after the objects it qualifies. Variables are indicated by question marks.

However, some planners cannot handle types. In fact, for almost every aspect of problem definition, there is some planner that cannot handle it. To cope with this issue, we borrowed an idea from the UCPOP language, namely to specify explicitly the requirements a planner would have to be satisfy in order to handle this domain. That explains the field `(:requirements :adl)` that appears in the domain definition. If a planner can't handle the `:adl` package of requirements, then it can issue a warning when it sees this flag.

After the `:requirements` specification, the next field defines the `:types` that are specific to this domain. (Types like `object` and `integer` are inherited by all domains.) Then there is a list of `:predicates`, each of which is given with its argument types.

There are four actions in this domain. Each is defined by giving a *precondition,* which must be true for the action to be feasible, and an *effect,* which specifies what happens when the action is executed. The effect is typically a conjunction. A conjunct of the form `(not` p`)` means that p becomes false. A conjunct of the form `(when` c e`)` means that effect e happens only if condition c is true before the action occurs. (This is a "context-dependent effect.") A conjunct of the form `(forall (`—*vars*—`)` e`)` means that e happens for every instance of the variables.

So the `fly-airplane` action is defined as follows: You can fly an airplane from airport A to airport B if the airplane is at A; you can't fly any other vehicle between any other types of location. The effect of flying is that the airplane is at B, and no longer at A. Furthermore, everything that is in the airplane is also at B, and no longer at A. Note that we have to specify both that an object is at B and that it is no longer at A.

For comparison, Tables 2 and 3 gives the same domain with no extra requirements at all. We call this baseline notation the *Strips notation* because it is essentially the same as the notation used by the Strips planner [6]. Here are the changes required to transform the ADL version into the Strips version:

1. Types are replaced by unary predicates. This ultimately requires splitting the actions `load` and `unload` into two versions each, one dealing with trucks and the other with airplanes.

2. Context-dependent effects ("`whens`") must be eliminated. In this case, the semantics of actions change. In the ADL version, an object is at the destination as soon as its vehicle moves there. In the Strips version, it is at the destination only when it is unloaded.

We will return to this issue of domain notation in Section 3.

Numbers are built in to all PDDL domains, but only those declaring requirement `:expression-evaluation` can have arithmetic expressions such as `(+ ?x 1)`. These occur in special contexts such as `(eval` e v`)`, where e is an arithmetic expression, and v must unify with its value. If e evaluates to a Boolean, then `(test` e`)` succeeds if and only if the value of e is `true`. `(equation` e_1 e_2`)` tries to bind variables in such a way as to make e_1 and e_2 evaluate to the same value. For instance, the goal `(equation (+ ?x 1) 3)` can be satisfied by binding `?x` to 2. Currently that's about the only pattern that implementations are required to handle.

For example, in defining a "grid world" in which a robot can move between locations with integer-valued coordinates, we can use these facilities to specify

```
(define (domain logistics-adl)
  (:requirements :adl)
  (:types physobj - object
          obj vehicle - physobj
          truck airplane - vehicle
          location city - object
          airport - location)
  (:predicates (at ?x - physobj ?l - location)
               (in ?x - obj ?t - vehicle)
               (in-city ?l - location ?c - city)
               (loaded ?x - physobj))   ; ?x is loaded on a vehicle

  (:action load
             :parameters (?obj ?veh ?loc)
             :precondition (and (vehicle ?veh)
                                (location ?loc)
                                (at ?obj ?loc)
                                (at ?veh ?loc)
                                (not (loaded ?obj)))
             :effect (and (in ?obj ?veh)
                          (loaded ?obj)))
  (:action unload
             :parameters (?obj ?veh ?loc)
             :precondition (and (vehicle ?veh)
                                (location ?loc)
                                (in ?obj ?veh)
                                (at ?veh ?loc))
             :effect (and (not (in ?obj ?veh))
                          (not (loaded ?obj))))
  (:action drive-truck
             :parameters (?truck - truck ?loc-from ?loc-to - location
                          ?city - city)
             :precondition (and (at ?truck ?loc-from)
                                (in-city ?loc-from ?city)
                                (in-city ?loc-to ?city))
             :effect  (and (at ?truck ?loc-to)
                            (not (at ?truck ?loc-from))
                            (forall (?x - obj)
                               (when (and (in ?x ?truck))
                                     (and (not (at ?x ?loc-from))
                                          (at ?x ?loc-to))))))
  (:action fly-airplane
   :parameters (?plane - airplane ?loc-from ?loc-to - airport)
   :precondition (and (at ?plane ?loc-from) )
   :effect  (and (at ?plane ?loc-to)
                (not (at ?plane ?loc-from))
                (forall (?x - obj)
                   (when (and (in ?x ?plane))
                         (and (not (at ?x ?loc-from))
                              (at ?x ?loc-to)))))))
```

Table 1: The Logistics Domain — ADL Version

```
(define (domain logistics-strips)
  (:requirements :strips)
  (:predicates (obj ?obj)
               (truck ?truck)
               (location ?loc)
               (airplane ?airplane)
               (city ?city)
               (airport ?airport)
               (at ?obj ?loc)
               (in ?obj1 ?obj2)
               (in-city ?obj ?city))
  (:action load-truck
           :parameters (?obj ?truck ?loc)
           :precondition (and (obj ?obj)
                              (truck ?truck)
                              (location ?loc)
                              (at ?truck ?loc)
                              (at ?obj ?loc))
           :effect (and (not (at ?obj ?loc))
                        (in ?obj ?truck)))
  (:action load-airplane
           :parameters (?obj ?airplane ?loc)
           :precondition (and (obj ?obj)
                              (airplane ?airplane)
                              (location ?loc)
                              (at ?obj ?loc)
                              (at ?airplane ?loc))
           :effect (and (not (at ?obj ?loc))
                        (in ?obj ?airplane)))

  (:action unload-truck
           :parameters (?obj ?truck ?loc)
           :precondition (and (obj ?obj)
                              (truck ?truck)
                              (location ?loc)
                              (at ?truck ?loc)
                              (in ?obj ?truck))
           :effect (and (not (in ?obj ?truck))
                        (at ?obj ?loc)))

  (:action unload-airplane
           :parameters (?obj ?airplane ?loc)
           :precondition (and (obj ?obj)
                              (airplane ?airplane)
                              (LOCATION ?loc)
                              (in ?obj ?airplane)
                              (at ?airplane ?loc))
           :effect (and (not (in ?obj ?airplane))
                        (at ?obj ?loc)))
  ...
```

Table 2: The Logistics Domain — Strips Version, Part 1

```
  ...
  (:action drive-truck
           :parameters (?truck ?loc-from ?loc-to ?city)
           :precondition (and (truck ?truck)
                              (location ?loc-from)
                              (location ?loc-to)
                              (city ?city)
                              (at ?truck ?loc-from)
                              (in-city ?loc-from ?city)
                              (in-city ?loc-to ?city))
           :effect (and (not (at ?truck ?loc-from))
                        (at ?truck ?loc-to)))
  (:action fly-airplane
           :parameters (?airplane ?loc-from ?loc-to)
           :precondition (and (airplane ?airplane)
                              (airport ?loc-from)
                              (airport ?loc-to)
                              (at ?airplane ?loc-from))
           :effect (and (not (at ?airplane ?loc-from))
                        (at ?airplane ?loc-to)))
)
```

Table 3: The Logistics Domain — Strips Version, Part 2

```
(define (domain jug-pouring)
   (:requirements :typing :fluents)
   (:types jug)
   (:functors
        (amount ?j - jug)
        (capacity ?j - jug)
        - (fluent number))
   (:action empty
       :parameters (?jug1 ?jug2 - jug)
       :precondition (fluent-test
                        (>= (- (capacity ?jug2) (amount ?jug2))
                            (amount ?jug1)))
       :effect (and (change (amount ?jug1)
                            0)
                    (change (amount ?jug2)
                            (+ (amount ?jug2)
                               (amount ?jug1)))))
   ...)
```

Table 4: The Jug-Pouring Domain

what it means for two coordinates to be adjacent. Here is a piece of that specification:

```
   (:axiom
      :vars (?i ?j ?i1 - integer)
      :implies (adjacent ?i ?j ?i1 ?j right)
      :context (and (equation (+ ?i 1) ?i1)
                    (legal_coord ?i)
                    (legal_coord ?i1)))
```

This example also illustrates PDDL's ability to represent axioms that delimit the meanings of symbols, such as `adjacent` and `right`. (There are three other axioms, for `left`, `up`, and `down`.)

There is also a notion of a term whose value changes in different situations, called *fluents*, following [9]. This feature is especially useful in domains where quantities can change. The classical domain in which water can be poured from jug to jug might be defined as in Table 4. The (`change` f e) says that the value of fluent f changes to the value of e before the change. The `:functors` declaration is used to add new function-defining symbols. Currently it can be used only to define new fluent constructors. So the type of `amount` is `(fluent number)`, meaning that `(amount` x`)` is a number that varies from situation to situation.

Fluents are a natural generalization of traditional effects; instead of specifying how truth values change, they allow specification of how terms change.

Without fluents, one could make the same definitions, but their meaning would be less clear. The precondition of `empty` would be

```
(and (amount-in ?jug1 ?a1)
     (amount-in ?jug2 ?a2)
     (capacity ?jug2 ?c2)
     (test >= (- ?c2 ?a2) ?a1))
```

and the effect would be

```
(and (not (amount-in ?jug1 ?a1))
     (amount-in ?jug1 0)
     (not (amount-in ?jug2 ?a2))
     (amount-in ?jug2 (+ ?a2 ?a1)))
```

While this formulation causes little trouble in inferring the effects of a known action, it is difficult to use to constrain the arguments and preconditions of an `empty` action given a goal such as `(and (amount-in jugB ?x) (> ?x 5))`. The problem is that there may be many ways to make the first conjunct true, but the result is to leave some number of gallons in `jugB`. That number is either greater than 5 or it isn't; there's no way to cause it to become bigger. Expressing the goal as `(> (amount jugB) 5)` is much more perspicuous.

To support hierarchical planning, PDDL allows actions to be defined that are carried out by executing a structure of more primitive actions. Our nuclear-reactor example appears in Table 5. In English: "To restart reactor `?r`, make sure it is not already running and not melting down, then open the two valves, toggle the switch, and close one of the valves, in that order." The `:vars` clause is used to declare local variables that are inconvenient to consider as parameters of the action. Every reactor is supposed to have one auxiliary valve, one main valve, and one main switch, so there is no need to name them as part of the action.

This definition is misleading in that it appears that there is only one way to expand an action. In general it is possible to specify multiple methods for an action expansion, as in the fragment shown in Table 6 from a domain involving shipping packages from one place to another. Here there are two methods for carrying out (`ship` x l_1 l_2). The first, carrying it in a plane, works only if x is a piece of mail. The second, using a truck, works for any cargo item. The expressions (`in-context` A `:precondition` p) means that p must be true before this occurrence of A in order for the plan to be valid.

In addition to defining domains, PDDL allows for the definition of problems. Table 7 gives an example in the Logistics domain. This is one of the examples used in the competition. A problem is defined as a domain, a set of :objects, an initial situation, and a goal pattern to be made true. In domains with action expansions, a problem can have an `:expansion` field, like this:

```
(define (problem trans-1)
   (:domain transportation)
   (:init .....)
```

```
(:action restart
    :parameters (?r - reactor)
    :vars (?valve1 ?valve2 - valve ?switch1 - switch)
    :precondition (and (not (running ?r))
                       (not (melting-down ?r))
                       (aux-valve ?r ?valve1)
                       (main-valve ?r ?valve2)
                       (main-switch ?r ?switch))
    :effect (running ?r)
    :expansion (series (verify-valves-shut ?r)
                       (parallel (open ?valve1)
                                 (open ?valve2))
                       (toggle ?switch)
                       (close ?valve1)))
```

Table 5: The Nuclear-Reactor Action

```
(:action ship
    :parameters (?pkg - cargo ?orig ?dest - location)
    :precondition (at ?pkg ?orig)
    :effect (at ?pkg ?dest)
    :expansion :methods)
(:method ship
    :parameters (?pkg - mail ?orig ?dest - location)
    :expansion (forsome (?p - airplane)
                 (series (in-context
                             (load ?pkg ?p)
                             :precondition (at ?p ?orig))
                         (fly ?p ?dest)
                         (unload ?pkg ?p))))
(:method ship
    :parameters (?pkg - cargo ?orig ?dest - location)
    :expansion (forsome (?tr - truck)
                 (series (in-context
                             (load ?pkg ?tr)
                             :precondition (at ?tr ?orig))
                         (drive ?tr ?dest)
                         (unload ?pkg ?tr))))
```

Table 6: The Shipping Action

```
(:goal (at truck3 detroit))
(:expansion (ship pkg13 cincinnati)))
```

Here the planner must find a way to carry out the action `(ship pkg13 cincinnati)` in such a way that `(at truck3 detroit)` when it is done.

Some of the other features of PDDL:

- Domains can include numerical parameters, such as the maximum coordinate in the grid world.
- It allows the specification of timelessly true propositions, i.e., facts that are present in all situations (thus saving having to enter them in all problem definitions).
- It allows one domain to be specified as a descendant of one or more alternative domains, so that it inherits types, axioms, actions, etc.
- It allows several problems to share an initial situation, which need be written only once. One initial situation can be defined in terms of small changes to another.
- Action expansions can include simple iterations, arbitrary acylic structures of actions, specification of conditions to be maintained true for some period during the plan, and more.

In addition to defining the language, we felt it was important to implement a syntax checker and a solution checker. The syntax checker could verify that domains submitted by others were valid PDDL, and ensure that no feature was used unless it was declared as a requirement. It could also count the amount of advice that was given. To make this possible, we required all planner-specific annotations to be indicated by a special flag. The syntax checker could measure the size of these annotations, and otherwise ignore them.

The other key piece of software was a solution checker. For the competition we wanted to be able to generate random problems. We anticipated not even knowing, for many of the problems, whether they had solutions or not. Some of the problems might have several solutions, some which might be quite long and involved. Having to check by hand if a solution was valid would be tedious and prone to error. We decided to automate it.

Doing so turned out to be harder than anticipated, for two reasons. The first is that, to our knowledge, no one has ever written a solution checker for a hierarchical planner, which you may find surprising. The reason is that most hierarchical planners do not treat prefabricated plans as part of the problem specification, but as advice on how to solve problems. Once an action sequence has been found, the hierarchical superstructure can be dropped, and the action sequence can be checked as though it had been found without the use of canned plans. Checking an action sequence is easy: Just do a little deduction to verify that every action in the sequence is feasible at the point where it is to be executed, and that the goal is true in the situation that results from executing the last action.

```
(define (problem log-x-2)
   (:domain logistics-adl)
   (:objects package5 package4 package3 package2 package1 - obj
             city10 city9 city8 city7 city6 city5 city4 city3 city2 city1
             - city
             truck10 truck9 truck8 truck7 truck6 truck5 truck4 truck3 truck2
             truck1 - truck
             plane4 plane3 plane2 plane1 - airplane
             city10-1 city9-1 city8-1 city7-1 city6-1 city5-1 city4-1
             city3-1 city2-1 city1-1 - location
             city10-2 city9-2 city8-2 city7-2 city6-2 city5-2 city4-2
             city3-2 city2-2 city1-2 - airport)
   (:init (in-city city10-2 city10) (in-city city10-1 city10)
          (in-city city9-2 city9) (in-city city9-1 city9)
          (in-city city8-2 city8) (in-city city8-1 city8)
          (in-city city7-2 city7) (in-city city7-1 city7)
          (in-city city6-2 city6) (in-city city6-1 city6)
          (in-city city5-2 city5) (in-city city5-1 city5)
          (in-city city4-2 city4) (in-city city4-1 city4)
          (in-city city3-2 city3) (in-city city3-1 city3)
          (in-city city2-2 city2) (in-city city2-1 city2)
          (in-city city1-2 city1) (in-city city1-1 city1)
          (at plane4 city3-2)
          (at plane3 city7-2) (at plane2 city3-2)
          (at plane1 city6-2) (at truck10 city10-1)
          (at truck9 city9-1) (at truck8 city8-1)
          (at truck7 city7-1) (at truck6 city6-1)
          (at truck5 city5-1) (at truck4 city4-1)
          (at truck3 city3-1) (at truck2 city2-1)
          (at truck1 city1-1) (at package5 city1-2)
          (at package4 city7-2) (at package3 city3-2)
          (at package2 city10-1) (at package1 city2-2))
   (:goal (and (at package5 city4-2)
               (at package4 city6-1)
               (at package3 city1-1)
               (at package2 city9-2)
               (at package1 city3-1))))
```

Table 7: The Logistics Problem LOG-X-2

Now suppose you add a serious requirement that the action sequence not just be legal, but also instantiate the `:expansion` given as part of the problem definition. The result is to superimpose a "parsing" problem on top of the standard deductive problem. That is, the solution checker must find a way to group the actions into a hierarchical structure so as to instantiate the given expansion, in a way analogous to the way a natural-language parser groups words into phrases. However, the problem is much more difficult, for several reasons. PDDL allows quantifiers in expansions, of the form `(forsome` v A`)` and `(foreach` v C A`)`. These occur in an action sequence if the right kind of instances of the action expansion A occur. In the case of a `forsome`, there must be one instance; in the case of a `foreach`, there must be a set of instances satisfying condition C. In addition, two action expansions could, unless constrained otherwise, be interleaved in an arbitrary order, and the same primitive action could occur as a part of more than one complex action.

It soon became clear that the problem of solution checking was going to be intractable unless the checker was given some hints. A solution to a problem with hierarchical expansions was going to have to include a specification of exactly which higher-level actions occurred where. Even with this change, the algorithm took a long time to develop, and it never was completely debugged.

Fortunately, or unfortunately, the lack of a stable algorithm for checking solutions never became a problem because no contestants appeared for this part of the competition. We corresponded with several potential entrants, but none of them got over the hurdles in the way of taking part. The main problem was, we believe, that the semantics of hierarchical planning have never been clarified to the point where everyone in this area can be said to be working on the same problem. Our attempt to create a "lowest common denominator" notation succeeded only in creating a new notation that matched no one's expectations. In addition, the hierarchical planning community is used to thinking of library plans as advice structures, which was a drastic departure from our assumption that the basic content of the plan library contained no advice, only "physics."

Trying to make this assumption actually work was extremely difficult. The problem is that no one has ever figured out how to reconcile the semantics of hierarchical plans with the semantics of primitive actions. Ordinary action sequences satisfy a straightforward *compositionality* property: If you know the preconditions and effects of two sequences of actions S_1 and S_2, then you can infer the preconditions and effects of S_1 followed by S_2. Hierarchical plans do not have this property, at least not obviously.

To take a simple example, consider the action `restart` , described in Table 5, for restarting a nuclear reactor. The action sequence ⟨`(verify-valves-shut r2), (open v30)`⟩ does not by itself restart the reactor, and neither does the action sequence ⟨`(open v29), (toggle sw53), (close v28)`⟩, but the two together do, assuming that all the relevant preconditions are satisfied. In other words, a "conditional effect" of the second sequence is to restart the reactor, in a situation where the first sequence has (just? recently?) been executed. Conditional effects are not unusual in classical planning [12], but they normally take the form of an effect that becomes true if and only if a certain *secondary* *precondition* was true before the action. This is the job of the `when` effect clause in PDDL. With hierarchical plans, we get a new kind of implicit precondition, that a certain standard plan be "in progress."

Suppose that an action sequence looks like ⟨`(open v29)`, `(toggle sw42)`, `(close v28)`⟩. Is it legal? Does it cause `(running r2)` to become true if `v29` is the auxiliary valve of `r2` and `sw42` is its main switch, and so forth? One might think that the answer is "Obviously not," because two actions are missing, namely `(verify-valves-shut r2)` and `(open v31)`, assuming `v31` is the main valve of `r2`. But to make this inference requires one to assume that these actions did *not* occur before the action sequence we explicitly mentioned.

As it turned out, in the end these complexities did not affect the actual competition. We describe them in such detail to save future researchers from rediscovering them.

3 The Contest

The competition took place in June, 1998, but the contestants spent several months preparing for it. Each of them had to alter the front end of their system to accept problems expressed in PDDL. Because the language was brand-new, this was an iterative process, in which changes to the notation were suggested and sometimes incorporated before the problems were specified.

Even more important, the form of the contest had to be worked out, and several sample problems had to be released, in order to give the contestants a clue about what their planners had to be capable of. A repository of problems was begun at Yale, using as a nucleus the repository developed by the UCPOP group at the University of Washington. Contestants were invited to submit new problems, and several did. In addition, some new domains were invented by the keeper of the repository, me.

3.1 Bargaining over Expressiveness

Over the six months leading up to the actual competition in June, 1998, there was an intricate negotiation involving the committee and the community of potential contestants. The committee wanted to encourage the research community to try new things; the community wanted the committee to focus on the areas their planners did well in.

In the case of hierarchical planners this tension proved fatal. The researchers with hierarchical planners lost interest rapidly as it became clear how great the distance was between PDDL and the kind of input their planners expected. Many of the researchers in this community think of their planners as a cross between a programming language and a knowledge-acquisition system. They have developed elaborate notations for capturing domain knowledge in the form of rules that push the planner toward particular kinds of solution. Unfortunately, PDDL defines all such notations as advice. To adapt these systems to PDDL would require factoring their input into two parts: a physics part that is identical for all planners, and an advice part that controls how the planner reacts to the physics. The difficulty of doing this separation under the time constraints proved to be insurmountable. After a few exploratory conversations all the hierarchical-planning researchers dropped out.

In the case of classical planning the committee assumed at first that the ADL track was where most of the interest would be. ADL had been around since the mid-1980s, and several existing systems had been able to handle problems expressed in that format. However, much of the progress on planning algorithms in the 1990s has been based on what might be called "propositional planning," in which variables are eliminated from planning problems by generating up front every instance of every term that might be needed. With the variables gone, all the machinery for matching literals and recording "codesignation constraints" [4] is not needed, and the search can focus on constraints among action terms and atomic formulas. The search process is simpler, and can afford to explore a lot more possibilities. The resulting algorithms offer a significant improvement over older approaches in many cases.

Unfortunately, this power has been attained by sacrificing some expressivity. The propositional-planning researchers have focused on the Strips notation for the time being, and their planners lack the ability to handle problems involving numbers, nonatomic terms, and quantifiers. Some of them also have trouble with context-dependent effects, in which the effects of an action depend on the circumstances in which it is executed. These may sound like serious limitations, but in many cases one can work around them, at the cost of using cumbersome notational tricks.

For example, one of the domains we came up with for the competition was called the "Mystery" domain. We called it that to conceal its underlying structure and make it harder to give planners advice about it. The domain actually concerned a transportation network through which vehicles could move carrying cargoes. A vehicle could move from one node to a neighboring node if there was fuel at the originating node. In PDDL:

```
(:action move
    :parameters (?v - vehicle ?n1 ?n2 - node)
    :precondition (and (loc ?v ?n1)
                       (conn ?n1 ?n2)
                       (fluent-test (> (fuel ?n1) 0)))
    :effect (and (not (loc ?v ?n1))
                 (loc ?v ?n2)
                 (change (fuel ?n1)
                         (- (fuel ?n1) 1))))
```

In English: you can move a vehicle from node 1 to node 2 if the vehicle is at node 1, node 1 is connected to node 2, and there is a nonzero amount of fuel at node 1. The effect of the move is for the vehicle to be at node 2, and for there to be one less unit of fuel at node 1.

There are two problems with this action definition: It involves numbers, and it involves a context-dependent effect (the amount of fuel afterward depends on the amount of fuel before). In PDDL, a term like `(fuel` n`)` defines a *fluent,* a term whose value changes from node to node. The notation `(change` f a`)` means that the value of fluent f after the action is equal to the value of expression a before the action.

It may seem as if this domain were simply off limits to any planner that can't handle numbers or context-dependent effects, but in fact there are ways to work around these limitations. Fuel amounts start off as nonnegative integers, never change except to become smaller by 1, and never become negative, so only a predictable set of natural numbers will occur in a given problem. Hence for every problem we can supply a set of constants `num0`, `num1`, ..., `num`K, where K is the largest number that occurs in the problem statement; and we can include among the initial conditions

```
(:init (just-less num0 num1)
       (just-less num1 num2)
       . . .
       (just-less num9 num10)
       ...)
```

(in the case where $K = 10$). These constants are declared as "pseudo-numbers":

```
(:objects num0 num1 . . . num10 - pseudo-number)
```

That eliminates the numbers; the next step is to eliminate the context-dependent effect. For this we use the old trick of adding arguments to the action. With this change our action definition becomes

```
(:action move
    :parameters (?v - vehicle ?n1 ?n2 - node
                 ?f1 ?f2 - pseudo-number)
    :precondition (and (loc ?v ?n1)
                       (conn ?n1 ?n2)
                       (fuel-at ?n1 ?f1)
                       (just-less ?f2 ?f1))
    :effect (and (not (loc ?v ?n1))
                 (loc ?v ?n2)
                 (not (fuel-at ?n1 ?f1))
                 (fuel-at ?n1 ?f2)))
```

Suppose `veh29` is at `node101`, which has 3 units of fuel. Instead of saying, e.g., "The action `(move veh29 node101 node63)` changes the fuel at `node101` from 3 to 2," we say, "The action `(move veh29 node101 node63 num3 num2)` is the only feasible action of form `(move veh29 node101 node63 ...)`."

It is somewhat discouraging that after thirty years of research we are back to the notational restrictions we started with. However, it did have one benefit. We wanted to disguise the "Mystery" domain, and all of this verbosity helped do that. We labeled nodes as "foods," vehicles as "pleasures," and cargo objects as "emotions." Instead of a single sort of pseudo-number, we introduced one, called "provinces," for fuel and another, called "planets," for space on vehicles. `(loc` v n`)` became `(craves` v n`)`; `(conn` n_1 n_2`)` became `(eats` n_1 n_2`)`. `(fuel` v k`)` became `(local` v k`)`. The `just-less` predicate for numbers became `(attacks` k_1 k_2`)`. Moving was called "feasting." So in the Strips style our action definition becomes:

```
(:action feast
    :parameters (?v ?n1 ?n2 ?f1 ?f2)
    :precondition (and (craves ?v ?n1)
                       (food ?n1)
                       (pleasure ?v)
                       (eats ?n1 ?n2)
                       (food ?n2)
                       (locale ?n1 ?f2)
                       (attacks ?f1 ?f2))
    :effect (and (not (craves ?v ?n1))
                 (craves ?v ?n2)
                 (not (locale ?n1 ?f2))
                 (locale ?n1 ?f1)))
```

There was also an ADL version, albeit without numbers, in which types and context-dependent effects were allowed. In that version, the action was defined as

```
(:action feast
    :parameters (?v - pleasure ?n1 ?n2 - food)
    :vars (?f1 ?f2 - province)
    :precondition (and (craves ?v ?n1)
                       (eats ?n1 ?n2)
                       (locale ?n1 ?f2)
                       (attacks ?f1 ?f2))
    :effect (and (not (craves ?v ?n1))
                 (craves ?v ?n2)
                 (not (locale ?n1 ?f2))
                 (locale ?n1 ?f1)))
```

3.2 Participants

As a result of bargaining between the committee and the contestants, we arrived at a Strips track and ADL track, neither of which could handle problems with numbers. By April of 1998, we had two contestants who were planning to enter the ADL track, and eight who were planning to enter the Strips track. No one wanted to enter the hierarchical-planning track, and the other track had never gotten off the ground. Nonetheless, we were happy with what we had. The contestants were putting a tremendous amount of work into altering their planners to take the PDDL notation. Unfortunately, for some of them the work was just too much, and three dropped out in the weeks leading up to the competition. The final participants in the Strips track were

- IPP (Jana Köhler, University of Freiburg, Germany)
- Blackbox (Henry Kautz and Bart Selman, AT&T Labs and Cornell University, US)
- HSP (Hector Geffner and Blai Bonet, Simon Bolivar University, Venezuela)
- STAN (Derek Long and Maria Fox, Durham University, UK)

The two participants in the ADL track were

- Köhler's IPP and
- SGP (Corin Anderson, University of Washington, US)

All the planners were written in C/C++ except for SGP, which was written in Lisp.

As explained in the companion paper, all of these systems except HSP were based to some extent on the Graphplan algorithm of [3]. HSP was based on heuristic search guided by means-ends analysis. In addition, Blackbox used satisfiability testing. All of the systems avoided repeated variable substitution by generating all required instances of propositions and action terms at the beginning. This lack of diversity in current research directions in classical planning means either that Graphplan on variable-free terms really is the best approach to planning, or that the summer of 1998 happened to coincide with the peak of a particularly intense fad. What was particularly striking was the complete absence of partial-order, or "nonlinear," planning [13]. A few years ago many people thought that the superiority of partial-order techniques had been proven conclusively [2]. It seems doubtful that the arguments in its favor were *all* wrong, and it would be interesting to see partial-order planners compete in future competitions.

3.3 Scoring

In parallel with the design of domains, we were also designing the scoring mechanism. This proved to be a difficult challenge, one that we never really solved. At first we thought the biggest issue was going to be how to penalize a planning system for taking advice. Some members of the community feel that there is nothing wrong with advice; it was even suggested that a planner be *rewarded* for being able to take it. However, most people agreed that if planner A requires a lot more advice than planner B to solve a problem, B should win if it does almost as well as A.

Here is the scoring algorithm we proposed:

The basic idea was to give each planner j a score on problem i equal to

$$(N_i - R_{ij})W_i$$

where N_i is the number of planners competing on problem i; R_{ij} is the rank of planner j on problem i (0 for best program, $N-1$ for worst, as explained below); and W_i is the difficulty of a problem, defined as

$$W_i = \frac{\text{median}_j T_{ij}}{\sum_m \text{median}_n T_{mn}}$$

where T_{bl} is the time taken by planner l on problem b.

Here is our method for computing R_{ij}: Rank all planners lexicographically as follows:

- Most important dimension: Correctness. There are two possible outcomes for planner j on problem i, in order of decreasing winnitude: either it stops and reports a correct answer, or it doesn't. In other words, either it

  1. Prints a correct solution or returns "NO SOLUTION" when there isn't one.
  2. Or it prints an incorrect solution; or returns "NO SOLUTION" when there is one; or never stops and has to be stopped by hand.

- Second dimension: Advice. Define

  $$A_{ij} = a_{ij} + A_{D_i,j}/N(D_i)$$

  where a_{ij} is the size of the advice given to planner j for problem i; A_{D_j} is the size of the advice given to planner j for domain D (D_i is the domain of problem i); and $N(D)$ is the number of problems in domain D.

  We planned to measure the size of a piece of advice by counting the number of symbols in it.

- Third dimension: Performance. If a problem has no solution, this is just the measured CPU time of planner j on problem i, or T_{ij}. If it has a solution, and planner j finds a solution, then we will replace T_{ij} with $T_{ij}(L_{ij})^h$, where L_{ij} is the length of the solution and $h = 0.5$. Length is defined as number of steps, regardless of whether some could be executed in parallel. (If $L = 0$, we will treat the solution as of length 1.)

Comments:

1. The idea was to take solution length into account, but to discount it so that it broke a tie between planners only if they had comparable solution times. If planner P_1 is 10 times slower than planner P_2, it would have to produce a plan 100 times shorter to win. If P_2 produces a plan twice as long, it must run in 70% of the time P_1 takes to beat it. This is to reflect the classical presupposition that existence of a plan is more important than its size. (Making the exponent h bigger would make length more relevant.)

2. If a planner required advice, it could never beat a planner that solves the same problems with no advice. So it's worth giving a certain amount of advice only when you bet that no one will be able to solve the problem with less.

3. For some of the more difficult machine-generated problems, we may not really know if there is a solution. In that case, if no planner finds a solution, we will assume that "NO SOLUTION" is the correct answer. If a planner has to be stopped by hand, then it will be taken to have returned "NO SOLUTION" after the amount of run time it actually spent (as close as that can be estimated).

Unfortunately, this scoring function, in spite of its arcane complexity, failed to match everyone's judgement about what was to be measured, as we will discuss below. Also, it turned out that our preoccupation with advice was misplaced. None of the competitors ever used any advice at all. A few years ago almost every planner would have required a lot of advice, and it is remarkable how big a change had occurred.

3.4 Domains

In the last month before the competition, everyone involved put in a tremendous amount of work, making sure that every planner worked on the sample problems. Contestants were invited to contribute problem domains of their own. The idea was to allow each of them to benefit from their areas of strength by having at least one domain where they knew they would do well. Domains were submitted by the IPP group, the SGP group, and the Blackbox group.

The final lineup of domains was this:

1. The Mystery domain described above. It defined three actions, corresponding to loading something on a vehicle, unloading it, and moving the vehicle.

2. "Mystery-prime:" This is the mystery domain with one extra action, the ability to squirt a unit of fuel from any node to any other node, provided the originating node has at least two units. The contestants knew that a modified Mystery domain was coming, but did not actually see it until the first day of the competition.

3. Movie: In this domain, the goal is always the same (to have lots of snacks in order to watch a movie). There are seven actions, including `rewind-movie` and `get-chips`, but the number of constants increases with problem number. Some planners have combinatorial problems in such cases. This domain was created by Corin Anderson.

4. Gripper: Here a robot must move a set of balls from one room to another, being able to grip two balls at a time, one in each gripper. There are three

actions, `move`, `pick`, and `drop`. Most planners explore all possible combinations of balls in grippers, overlooking the fact that all combinations are equivalent, and giving rise to an unnecessary combinatorial explosion. (Contributed by Jana Köhler.)

5. Logistics: There are several cities, each containing several locations, some of which are airports. There are also trucks, which can drive within a single city, and airplanes, which can fly between airports. The goal is to get some packages from various locations to various new locations. (Created by Bart Selman and Henry Kautz, based on an earlier domain by Manuela Veloso.) Table 1 gives the complete ADL version of the logistics domain, developed by me from Selman and Kautz's Strips version.[1] The Strips version has six action definitions instead of four, because an action with context-dependent effects has to be split into different versions.

6. Grid: There is a square grid of locations. A robot can move one grid square at a time horizontally and vertically. If a square is locked, the robot can move to it only by unlocking it, which requires having a key of the same shape as the lock. The keys must be fetched, and may themselves be in locked locations. Only one object can be carried at a time. The goal is to get objects from various locations to various new locations. The ADL version of the domain has four actions, and the Strips version has five. (Created by Jana Köhler, based on an earlier domain of mine.)

7. Assembly: The goal is to assemble a complex object made out of subassemblies. There are four actions, `(commit` *resource assembly*`)`, `(release` *resource assembly*`)`, `(assemble` *part assembly*`)`, and `(remove` *part assembly*`)`. The sequence of steps must obey a given partial order. In addition, through poor engineering design, many subassemblies must be installed temporarily in one assembly, then removed and given a permanent home in another. There was no Strips version of this domain.

It would be pleasant if we could claim that these domains covered the entire range of what planners can handle; or that these domains represent approximations of real-world problems planners will eventually solve; or that within each domain the problems are typical. Unfortunately, we can make none of those claims. Two of the domains, Movie and Gripper, were submitted because problems in these domains were thought to be difficult for some planners to solve, even though the problems are easy for humans. The other domains were chosen because it seemed, based on experience and informal experimentation, that it was possible to create hard problems in them. However, as is now well known[8], it can be tricky to generate *random* problems that are hard. Randomly generated problems tend to be either extremely easy or impossible. (Some "impossible" problems are actually quite easy, because many programs can quickly verify that they are impossible.) The zone in between the subspace

[1] This is not precisely the version of the domain used in the competition; that version had an unimportant bug which has been removed.

of easy problems and the subspace of impossible problems has been compared to a "phase transition" in a physical system. Analyzing a domain to figure out where the phase transitions are is not easy, and we did not attempt it for any of the domains in the competition. As a result, some of the randomly generated problems are too easy, and some are too hard. However, it does seem that many are about right.

4 Results

The competition took place at Carnegie Mellon University, in June of 1998, at the same time as AIPS-98. We owe a debt of gratitude to the CMU staff, especially Bob McDivett, who got the computers running, and made sure that they were all identical. (The computers were 233MHz Pentium-based PC-compatibles, with 128 MBytes of primary memory, running the Linux operating system.) The contestants and I arrived early at the conference in order to get their systems up and running. The next few days were an intense but exhilarating effort. In the end we had to write quite a bit more code, and rethink our scoring function completely.

There were to be two rounds in the competition. The first was designed to allow contestants to get used to the environment and the problem domains. They would be allowed to run their programs several times and make changes in between. The programs that did best in Round 1 would be allowed to advance to Round 2, where the rules were more stringent. Some new domains would be introduced. Each planner could be run exactly once, with no tuning. Round 2 would take place in "real time," as the conference proceeded, with programs' performance announced as they finished.

Everything went reasonably smoothly through Round 1. We had a total of 170 problems, drawn from the Assembly, Gripper, Logistics, Movie, and Mystery domains. All 170 appeared in the ADL track; in the Strips track, the Assembly problems were omitted, leaving 140. Table 8 summarizes the data on problem sizes. The Grid domain was reserved for Round 2. Contestants worked through Monday, June 8, at 5 PM, when we declared Round 1 complete. On the ADL track, IPP outperformed SGP so convincingly that it was declared the winner. Both programs did well, but SGP was written in Lisp, and rarely matched the raw speed of the other systems, which were written in C or C++. We will display the results below.

The results for the Strips track were not at all clear. For one thing, we failed to anticipate that several of the contestants would simply not try to solve some of the problems. If their planner failed on almost all of the easiest 10 problems in a domain, they didn't see the point of letting it grind forever on the next 20. The scoring function as originally designed gave one point to a program that tried a problem, failed, and took longer than any other program that tried and failed. It gave zero to a program that didn't try.

An even worse problem was that one planner (STAN) spent an hour each on the more difficult Gripper problems before giving up on them. This was

Domain	*Number*	*Av. Obs*	*Av. Inits*	*Smallest*	*Largest*
Assembly	30	48	118	67	270
Gripper (ADL)	20	25	26	13	89
Gripper (Strips)		27	53	23	137
Logistics (ADL)	30	171	155	58	960
Logistics (Strips)		171	342	96	1470
Movie (ADL)	30	98	3	28	173
Movie (Strips)		98	99	51	341
Mystery (ADL)	30	44	82	46	233
Mystery(Strips)		44	126	64	317

For each domain, the number of problems, the average number of objects per problem, and the average number of "inits" (propositions true in the initial situation) are shown. The columns labeled *Smallest*, and *Largest* give the combined size (objects + inits) for the smallest problem in the domain and the largest. The Mystery-prime domains had exactly the same problems as the Mystery domain, and so are not listed separately here.

Table 8: Problem Sizes — Round 1

much longer than any other planner spent on any problem. Only two planners, STAN and HSP, tried the difficult Gripper problems, so the median time to solve them was quite large, and these Gripper problems ended up carrying a large fraction of the weight. HSP spent less time than STAN, and actually solved the problems (although not optimally), so it got a higher overall score on Round 1 than anyone else. The HSP team deserves credit for solving these problems, but it seems clear that the scoring function's judgement of their importance disagrees with intuition.

At the end of Round 1, therefore, all four contestants in the Strips track could argue that their systems had done well. A total of 88 problems had been solved by at least one planner. HSP solved more problems than any other system, and found the shortest solution more often. Blackbox had the shortest average time on problems it attempted, but IPP had the shortest solution time on more problems. STAN was in second place for shortest solution time, and second place for overall score.

The committee was unhappy with the holes that had been revealed in the scoring function. We tried to achieve a consensus on what to replace it with, and finally gave up. We decided to let all the programs advance to Round 2, measure their performance as well as possible, and let history judge who, if anyone, did the best. Once this decision was made, Round 2 was a lot of fun. We continued to observe the same pattern as in Round 1, that different planners succeeded in different ways.

For Round 2, we used the Grid, Logistics, and Mystery-prime domains, all in their Strips versions. There were a total of 15 problems, of which 12 were solved

Domain	*Number*	*Av. Obs*	*Av. Inits*	*Smallest*	*Largest*
Logistics (Strips)	5	33	67	63	159
Mprime (Strips)	5	40	106	82	237
Grid (Strips)	5	66	328	209	613

Table 9: Problem Sizes — Round 2

by at least one program. Table 9 shows the sizes of these problems. We tried to generate problems that the planning systems could be expected to handle in the time allotted.

Tables 10 and 11 summarize how hard the problems were that some planner could solve. For each domain we show the largest problem that any planner could solve, and the problem whose shortest known solution is longer than that of any other problem. These figures should not be taken too seriously. For one thing, the fact that a problem was not solved by any planner may mean that it has no solution; more on this below. In addition, the fact that no planner finds a short solution to a problem does not mean that there isn't one. To give a concrete idea of the performance of the planners, Table 7 contains the definition of problem STRIPS-LOG-X-2, which occurred during Round 1 of the competition. It mentions 49 objects and 68 inits, for a total size of 147. Both STAN and Blackbox found 32-step plans to solve this problem; HSP found a 44-step plan. Blackbox's plan is shown in Table 12.

Tables 13–15 give the results for both rounds. The planners are sorted in alphabetical order in each table. These results are not exactly the same as those were presented at the conference, because of some minor glitches that muddied the presentation. Three of IPP's solutions were checked and found to be wrong. It turned out that the reason for this was a trivial bug in the output printer, which caused all occurrences of one particular action to be garbled. In these tables we have counted these as successes.

There are two important caveats about these data:

1. We measure the length of the plan found by counting the total number of steps in it. However, for many of the planners this may not be the appropriate measure. Planners like Blackbox, IPP, and STAN find the plan with the shortest "parallel length," in which several steps are counted as taking one time unit if they occur as a substring at some point in the plan and the substring could have occurred in any order. The plan with the shortest parallel length may not be the plan with the least number of steps. Which of these numbers is a better measure of plan quality is not always obvious.

2. If no planner found a solution to a problem, it simply doesn't enter into our statistics. But in some cases some of the planners were able to prove there was no solution. We discuss this further below.

For further analysis of results, see the companion paper.

Domain	*Largest Solved*	*Plan Length*	*Solution Time*	*Longest Solution*	*Solution Time*
Gripper (ADL)	29	47	225730	47	225730
Gripper (Strips)	137	165	33210	165	33210
Logistics (ADL)	109	26	17400	26	17400
Logistics (Strips)	180	112	788914	112	788914
Movie (ADL)	173	7	50	—	—
Movie (Strips)	341	7	40	—	—
Mystery (ADL)	159	4	17810	13	9280
Mystery(Strips)	304	16	1789	16	789
Mprime (ADL)	131	10	24240	12	1960
Mprime (Strips)	214	4	7141	11	5214

For each domain, data are shown for the hardest problems solved. In the left side of the table, data are displayed for the largest problem solved. Call it B. "Largest solved" is the size of B, defined as the sum objects + inits. "Plan Length" is the length of the shortest solution of B found, and "Solution Time" the time the fastest planner took to find that solution. In the right side of the table, data are displayed for the problem whose shortest solution was longest. Call it L. "Longest Solution gives the length of the shortest solution of L found, and "Solution Time" is the time the fastest planner took to find that solution. Times are in milliseconds.

Table 10: Hardest Problems — Round 1

Domain	*Largest Solved*	*Plan Length*	*Solution Time*	*Longest Solution*	*Solution Time*
Logistics (Strips)	159	31	66170	59	170394
Mprime (Strips)	237	6	4991	7	2537
Grid (Strips)	209	14	2505	14	2505

This table is in the same format as table 10. Times are in milliseconds.

Table 11: Hardest Problems — Round 2

```
((load-airplane package4 plane3 city7-2)
 (load-truck package2 truck10 city10-1)
 (load-airplane package3 plane2 city3-2)
 (drive-truck truck1 city1-1 city1-2 city1)
 (fly-airplane plane4 city3-2 city2-2)
 (fly-airplane plane1 city6-2 city10-2)
 (drive-truck truck6 city6-1 city6-2 city6)
 (drive-truck truck3 city3-1 city3-2 city3)
 (load-airplane package1 plane4 city2-2)
 (drive-truck truck10 city10-1 city10-2 city10)
 (fly-airplane plane3 city7-2 city6-2)
 (fly-airplane plane2 city3-2 city1-2)
 (unload-airplane package4 plane3 city6-2)
 (unload-truck package2 truck10 city10-2)
 (fly-airplane plane4 city2-2 city3-2)
 (unload-airplane package3 plane2 city1-2)
 (load-airplane package5 plane2 city1-2)
 (fly-airplane plane2 city1-2 city4-2)
 (load-airplane package2 plane1 city10-2)
 (unload-airplane package1 plane4 city3-2)
 (load-truck package3 truck1 city1-2)
 (load-truck package4 truck6 city6-2)
 (drive-truck truck6 city6-2 city6-1 city6)
 (unload-airplane package5 plane2 city4-2)
 (fly-airplane plane1 city10-2 city9-2)
 (drive-truck truck1 city1-2 city1-1 city1)
 (load-truck package1 truck3 city3-2)
 (unload-airplane package2 plane1 city9-2)
 (drive-truck truck3 city3-2 city3-1 city3)
 (unload-truck package3 truck1 city1-1)
 (unload-truck package4 truck6 city6-1)
 (unload-truck package1 truck3 city3-1))
```

Table 12: Blackbox's Solution to the Problem of Table 7

Planner	*Av. Time*	*Problems Solved*	*Fastest*	*Shortest*
IPP	21396	69	68	68
SGP	14343	38	5	35

A total of 69 problems were solved by one or both of the planners; there was a tie for fastest planner 4 times. Times are in milliseconds.

Table 13: Results for Round 1 — ADL Track

Planner	*Av. Time*	*Problems Solved*	*Fastest*	*Shortest*
BLACKBOX	1498	63	16	55
HSP	35483	82	19	61
IPP	7408	63	29	49
STAN	55413	64	24	47

A total of 88 problems were solved by at least one planner. Times are in milliseconds.

Table 14: Results for Round 1 — Strips Track

Planner	*Av. Time*	*Problems Solved*	*Fastest*	*Shortest*
BLACKBOX	2464	8	3	6
HSP	25875	9	1	5
IPP	17375	11	3	8
STAN	1334	7	5	4

A total of 12 problems were solved by at least one planner. Times are in milliseconds.

Table 15: Results for Round 2 — Strips Track

5 Conclusions

The planning competition was a valuable exercise. For the first time researchers had to compete against each other on exactly the same problems. The PDDL notation made this possible, and hopefully it will continue to serve this role. The PDDL syntax checker and solution checker, as well as all the problems and results from the competition, can be found at

http://www.cs.yale.edu/ dvm

We encourage researchers to compare their planning systems against the programs that competed. However, the existence of this repository is only a first step toward a comprehensive set of benchmark problems for automated planners. We encourage others to submit candidate benchmarks to drew.mcdermott@yale.edu, or to the competition committee for the 2000 Planning Competition, chaired by Fahiem Bacchus (fbacchus@cs.toronto.edu).

The competition documented a dramatic increase in the speed of planning algorithms. Some of the problems in the competition had solutions of 30 steps or more, extracted from problems with dozens of propositions to deal with. Ten years ago planners required significant amounts of domain-specific advice in order to achieve performance like this; the current generation requires no advice at all.

To an extent this gain has been won by restricting the types of problems that can be worked on. Most of the planners could handle Strips-style problems and not much else. Some of these restrictions are only temporary, and we would urge the planning community to explore ways of removing them. However, the focus on classical planning, where perfect information is assumed, seems to be an intrinsic constraint on most planners currently being developed. There's nothing wrong with this focus, but if the current research is really leading to powerful algorithms, it will soon be time to show them working on realistic classical-planning problems. For instance, combining features of the Logistics and Mystery domains would get us close to real-world transportation planning with capacity constraints. Perhaps this is a reasonable target for the community to aim for.

There is a remarkable divergence of opinion on whether and how planners should take "advice", that is, domain- or problem-specific guidelines that are not strictly necessary in defining the domain, but constrain the search for plans. Some researchers feel that any need for advice is a weakness, while others think that there is unlikely to be a general-purpose planning algorithm that solves all realistic problems, so the issue is not whether domain-specific heuristics are necessary, but how easy or natural it is to tell a planner about them. In this competition the first camp decided the issue, probably because I am in that camp. But none of the competitors could resist the temptation to ask for a "little bit" of "trivial" advice. For instance, some of the propositional planners must set a bound on the length of a plan before doing a search. If the bound is unknown, they must search for it by starting with a short bound and extending it after each failure. A serious argument was made that this number should be given to the planner in advance. The anti-advice people succeeded in arguing that this one number constituted an enormous hint. (Among other things, it tells the planner that a solution exists.) In other cases, the disagreement became an impasse. For example, many hierarchical planners (especially SIPE [14]) give deductive rules a "procedural interpretation," so that they are used in only one direction. $P \supset Q$ is interpreted to mean that Q is *caused by* P, which then gets generalized to "Execute Q whenever P becomes true." This can be a very directed way of getting a planner to do something, and once you get used to thinking in these terms it's hard to recast problems as pure physics plus some extra advice.

We hope the competition will become a regular part of the AIPS conference, thereby continuing to exert pressure on the planning community. In particular, we hope the next competition deals with the following issues:

- There should be competitions involving hierarchical planning, but also run-time (reactive) planning, and decision-theoretic planning.

- The issue of plan quality should be addressed more carefully. Different problem domains have different definitions of optimality, and these should be made explicit in the domain or problem definitions. Plan length (sequential or parallel) is only a crude measure of plan cost; more realistic measures are needed. In many domains the problem of finding an optimal plan is much more difficult than the problem of finding a feasible one, and in those cases it may be desirable to give planners a big bonus for coming close to the optimum.

- Some planners can prove that a problem has no solution; others just work for a while and then declare that they can't solve it. The former should be rewarded. The problem is to verify that a planner's output is correct. In the case where it outputs a proposed solution as a sequence of steps, that sequence can be simulated to see if it is feasible and actually brings about the desired situation. The issue is what it should output in the case where it proves that there is no solution. One possibility is that it can produce a formal proof in first-order logic, in which case the solution checker would just be a proof checker. But most systems that prove there is no solution do not currently produce such a proof, and it would probably be a major pain to give them the ability to produce one.

- Many of the problems in the competition were produced by random problem generators. It is difficult to ensure that randomly generated problems are "interesting," i.e., not easy or impossible to solve. One way to get around this difficulty is to make problem design a more important part of the competition, and encourage participants (and others) to produce tricky problems.

- A more strenuous effort should be made to accommodate planners that require domain-specific advice. Ideally, a reward mechanism should be found that gives points for ease of advising, and more creative ways should be sought of combining PDDL with planner-specific advice.

Acknowledgements: Thanks to the members of the committee and the contestants for making this event possible. Thanks to the CMU staff for technical support. Thanks to Blai Bonet, Henry Kautz, Manuela Veloso, David Wilkins, and the referees for comments on this paper. Funding for the competition was provided by DARPA.

References

[1] Ronald C. Arkin. The 1997 Aaai mobile robot competition and exhibition. *AI Magazine* , 19(3):13–17, 1998.

[2] Anthony Barrett and Daniel S. Weld. Partial-order planning: evaluating possible efficiency gains. *Artificial Intelligence* , 67(1):71–112, 1994.

[3] Avrim L. Blum and Merrick L. Furst. Fast planning through planning graph analysis. In *Proc. Ijcai*, 1995.

[4] David Chapman. Planning for conjunctive goals. *Artificial Intelligence* , 32(3):333–377, 1987.

[5] Kutluhan Erol, Dana Nau, and James Hendler. Htn planning: complexity and expressivity. In *Proc. AAAI-94*, 1994. Seattle.

[6] Richard Fikes and Nils J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence 2*, pages 189–208, 1971.

[7] David M. Hart and Paul R. Cohen. Predicting and explaining success and task duration in the Phoenix planner. In *Proc. First Int'l. Conf. on AI Planning Systems*, pages 106–115, 1992.

[8] Scott Kirkpatrick and Bart Selman. Critical behavior in the satisfiability of random boolean expressions. *Science 1994*, 264(May):1297–1301, 1994.

[9] John McCarthy and Patrick Hayes. Some philosophical problems from the standpoint of artificial intelligence. In Bernard Meltzer and Donald Michie, editors, *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, 1969.

[10] MUC. *Proc. Third Message Understanding Conference.* Morgan Kauffman, 1991.

[11] MUC. *Proc. Fourth Message Understanding Conference.* Morgan Kauffman, 1992.

[12] Edwin Peter Dawson Pednault. Adl: Exploring the middle ground between Strips and the situation calculus. In *Proc. Conf. on Knowledge Representation and Reasoning* **1**, pages 324–332, 1989.

[13] Daniel Weld. An introduction to least-commitment planning. *AI Magazine*, 1994.

[14] David Wilkins. *Practical Planning: Extending the Classical AI Planning Paradigm.* Morgan Kaufmann Publishers, Inc, 1988.