The Transplan Domain

October 24, 1997

Drew McDermott
Yale University
mcdermott-drew@yale.edu

1 Introduction

The Transplan Domain is a hierarchical-planning domain for use in the AIPS-98 Planning Competition. When we say that the domain is hierarchical, we mean that part of the definition of a feasible action in the domain involves the use of some standard plans from a plan library. A problem statement may mention both that certain goals (states of affairs) are to be brought about, and that certain standard actions are to be carried out. In other respects, the domain is purely "classical," in that all actions are under the control of the planner, and there is perfect information about the initial situation and the effects of actions.

The domain is derived from the University of Maryland Translog Domain (Andrews et al. 1995), which was derived from a domain due to the Prodigy group at Carnegie Mellon University.

The files `transplan/domain.pddl` and `transplan/methods.pddl` contain a detailed and complete specification of the domain in PDDL, the Problem Domain Definition Language. (See file `pddl.ps`.) What follows is an overview to help understand the formal spec.

2 Overview

A problem in the Transplan domain is to transport one or more objects, called *packages*, to one or more *locations*. A location is a "small" place, such as a building, airport, fuel depot, etc. Locations occur in *cities*, and cities occur in *regions*. (Some locations are outside of cities.) Cities and locations are connected by *routes* of various kinds (air-routes, road-routes, and rail-routes). Locations are of two sorts: *transport-centers*, such as airports and train stations, and *non-transport-centers*, such as "street addresses," post offices, and depots.

Packages are carried in *vehicles*, which also are of various kinds (airplanes, trucks. trains, and traincars). Not every package can be transported in every kind of vehicle. The predicate `(can-carry` v p`)` is true of v and p if vehicle v can carry package p. This often depends on the "shape" of the package and the "specialty" of the vehicle. See `domain.pddl` for details.

Vehicles and locations can have *compartments*. For example, a vehicle with specialty `livestock-carrier` has three compartments: `gas-tank`, `water-tank`, and `cargo-area`. Compartments are labeled "generically," not as individuals. So you always have to refer to the owner of the compartment as well as the compartment itself. For example, to say that vehicle `truck13` has 100 liters of gasoline, write `(contains truck13 gas-tank 0.1)`. (Actually, that just says it has 0.1 cubic meter of something. To say it is gasoline, you also have to write `(contains-kind truck13 gas-tank fuel)`.)

The primitive actions in the domain allow you to load packages into vehicles, transfer liquids (fuel or water) between vehicles and storage tanks, and move vehicles from location to location. However, the primitives do not tell the whole story. Most of them cannot be used in isolation, but only as components of standard plans. (In PDDL notation, they have the field `:only-in-expansions=t`.) The standard plans for transporting objects are given in the file `methods.pddl`. They allow you to transport an object in one of the following two basic ways:

1. `(transport-direct` p l_o l_d`)`: `load` package p onto a vehicle, `move` the vehicle from l_o to l_d, and `unload` p from it. (This is only feasible if there is a direct route from l_o to l_d.)
2. `(transport-via-hub` p c_1 c_2`)`: Find a `hub` h, `transport-direct` from c_1 to h, then `transport-direct` from h to c_2. Note that h, c_1, and c_2 must all be transport centers.

As a sort of "macro," the action `(transport-between-tcenters` p c_1 c_2`)` means to do one of

- `(transport-direct` p c_1 c_2`)`
- `(transport-via-hub` p c_1 c_2`)`

in the case where c_1 and c_2 are transport centers.

The two basic transport methods, plus the `transport-between-tcenters` macro, can be used to implement `(transport` p l_o l_d`)` in one of the following combinations:

1. Just `transport-direct` from l_o to l_d.
2. If l_o and l_d are non-hub transport centers, then `transport-via-hub` from l_o to l_d.
3. If l_o is not a transport center, but l_d is, `transport-direct` from l_o to some transport center c , then `transport-between-tcenters` from c to l_d.
4. If l_o is a transport center, but l_d isn't, then `transport-between-tcenters` from l_o to some transport center c, then `transport-direct` from c to l_d.
5. If neither l_o nor l_d is a transport center, then find two transport centers c_1 and c_2, and `transport-direct` from l_o to c_1, then `transport-between-tcenters` from c_1 to c_2, then `transport-direct` from c_2 to l_d.

Note that this structure is not recursive. The longest possible sequence is of length four:

$$l_0 \longrightarrow c_1 \longrightarrow h \longrightarrow c_2 \longrightarrow l_d$$

i.e., from l_o to transport center c_1, then to hub h, then to transport center c_2, then to l_d. While this limits the search, it also limits the possibilities, because there may exist many routes from l_o to l_d, but if they don't fit the legal patterns they can't be used.

3 Vehicle Movements and Capacity Limitations

The primitive action for motion is `(move` v l_1 l_2 r`)`, where v is a vehicle, l_1 and l_2 are locations, and r is a route. The motion is possible only if one of the following is true:

1. the proposition `(connects` r l_1 l_2 d`)` is true for some distance d; i.e., l_1 and l_2 are connected by a direct route;
2. v is a truck, and `(connects` r c_1 c_2 d`)` is true, where c_1 and c_2 are the cities in which l_1 and l_2 are located; i.e., in reasoning about truck movements we can neglect intra-city motions.

The `move` action is used only inside expansions of `(achieve-vehicle-at` v l`)`, which can be strung together *ad lib*. Hence a vehicle can be gotten anywhere eventually if there is enough connectivity (as contrasted with package movement, which must be governed by `transport` schemas.)

Vehicles cannot hold an infinite amount. The predicate `(capacity` v c x`)` is true if x is the capacity in cubic meters of compartment c of vehicle or depot v. The sum of the packages loaded or liquid transferred to a compartment cannot exceed its capacity. (And, of course, you can never have a negative amount in a compartment.)

When a vehicle moves, it uses up fuel and time. Because we're in a classical-planning domain, we have the somewhat artificial convention that two vehicles cannot move simultaneously. (However, if the vehicle is a train, then all its cars move.)

Time passes only when a vehicle is in motion. If livestock are loaded into a vehicle, even a stationary one, they use up water at a rate that depends on the type of vehicle. (The water is in the `water-tank` compartment, and the animals are in the `cargo-area` compartment.) If they use up all their water, they die. Animals cannot be unloaded, and hence `transports` of animals cannot be completed, if the animals are dead.

The rates at which these changes occur are determined by the following predicates:

- `(fuel-rate` v r n`)`: n is the rate in liters/kg-km at which vehicle v burns fuel on route r. The "kg" in the denominator reflects the fact that if v is a train, the rate depends on its mass, i.e., the sum of the masses of its cars. (The mass of the cargo is not taken into account.)
- `(fuel-waste` v r w`)`: w is the fuel required for v to start and stop on route r.

- `(speed` v r s`)`: s is the speed in km/hr of vehicle v on route r.
- `(latency` v r l`)`: l is the time in hr for vehicle v to start and stop on route r.
- `(water-rate` v r`)`: r is the rate (liters/hr) per cubic meter of animal (!) that water is consumed when animals are in vehicle v.

As explained in Section 5, these capacities and rates will be ignored for some phases of the competition.

4 Packages

Objects to be transported are called "packages." The term encompasses some items that are not ordinarily thought of as packages, such as quantities of liquid or groups of animals. The terminology reflects the fact that packages must be treated as a unit. You can't break a group of animals into individual animals (let alone fractions of animals!) in order to cram them into cars that are already partially filled.

You can load two different packages into a compartment, and then extract them later, provided they are of the same kind, as specified by the predicate `(stuff` p s`)`, where p is a package and s is of type `kind-of-stuff`. The s argument is the same as in `(contains-kind` v c s`)`, specifying the kind of stuff in compartment c of vehicle or depot v. All "discrete" objects have `stuff items`. Liquid and granular objects are of various kinds; the only two defined as part of the domain are `fuel` and `water`.

As a consequence of these rules, you can put a shipment of refrigerators and a shipment of TV sets into a vehicle and get them out later. You could even put together and later separate a shipment of refrigerators and a herd of cows using the same vehicle (except that the current domain won't allow them both into vehicles of the same specialty). You can put two "packages" of fuel into a tank and take them out later, but you can't have a fuel package and a water package in the same tank at the same time.

The action for transferring liquids is called `(liquid-transfer` s c_s d c_d a`)`, where s is a source (vehicle or depot), d is a destination (vehicle or depot), c_s and c_d are the compartments involved, and a is the amount being transferred. This action can be used to load and unload "liquid packages," but it can also be used to transfer anonymous batches of fuel and water to keep vehicles and animals running.

A package is either `aboard` a vehicle or `at` a location, never both. If it is a liquid, it also has a container: `(container` p c`)` asserts that the "liquid package" p is in container c (of the vehicle or location where it currently resides).

5 Rules of the Competition

To compete in the competition, your program must be able to solve problems stated in PDDL, a manual for which appears in the file `pddl.ps`. (References to files are to files accessible at

```
ftp://ftp.cs.yale.edu/pub/mcdermott/software/pddl.tar.gz
```

which defines a group of files plus a subdirectory `transplan`. This manual is the file `transplan/doc.ps` in that subdirectory.)

Some contestants may not want to use PDDL as the domain definition. And some may want to use PDDL extended with advice of various kinds. See below.

In any case, the use of PDDL to state problems simply means that we will express problems in this format:

```
(define (situation transplan-world)
    (:domain transplan)
    (:init (package p1)
           (stuff p1 fuel)
           ...))

(define (problem prob-39)
    (:domain transplan)
    (:situation transplan-world)
    (:init (not (available ramp72))
           ...)
    (:expansion (parallel (transport p1 depot1 house22)
                          (transport p2 depot1 airport23)
                          ...)))
```

The solver must then be callable with the problem name (`prob-39`) as argument.

Because this is a hierarchical planning domain, a solution is not just a sequence of actions. Instead, it is a sequence of actions plus a *schema instantiation hierarchy*, which specifies how the actions in the sequence satisfy the legal expansions of the `transport` actions. That is, for each `transport` action in the problem statement, there must be an *action schema map* from the actions in the problem statement (and possibly other actions) to the solution sequence. An *action schema map* for an action term A consists of EITHER

- The position in the solution sequence of an action matching A;
- OR a *method* name from the plan library for A, plus:
  - for each step of the method, an action schema map for it;
  - for each goal of the method, two numbers specifying where in the solution sequence the goal becomes true and when it becomes false (or the end of the sequence if it never becoems false).

The map bottoms out in primitive actions that map to elements of the solution sequence.

A solution might look like:

```
(; Solution sequence:
 ((load-items p1 truck3)  ; interval 0 - 1
  (door-close truck3)     ;  1 - 2
  (move truck3 depot1 depot13 route66)  ; 2 - 3
  (door-open truck3)      ; 3 - 4
  (unload-items p1 truck3) ; 4 - 5
  (door-close truck3)) ; 5 - 6
 ; Action schema map:
 (((transport p1 depot1 depot13)
   :method transport-direct
   (; list of submaps
    ((transport-direct p1 depot1 depot13)  ; submap begins here
     :method just-do-it
     (((at truck3 depot1) :goal (0 2))
      ((load p1 truck3)
       :method load-items
       (((door-open truck3) :goal (0 1))
        ((load-items p1 truck3) :primitive (0 1))
        ((at p1 depot1) :goal (0 0))
        ((at truck3 depot1) :goal (0 2))
        ((aboard p1 truck3) :goal (1 4))
        ((at truck3 depot1) :goal (0 2))
        ((not (door-open truck3)) :goal (2 3))))
      ((at truck3 depot13)
       :goal (3 5))
      ((unload p1 truck3)
       :method unload-items
       (((aboard p1 truck3) :goal (1 4))
        ((at truck3 depot13) :goal (3 5))
        ((door-open truck3) :goal (4 5))
        ((unload-items p1 truck3) :primitive (4 5))
        ((not (door-open truck3)) :goal (6 6))))))))
  ((achieve-vehicle-at truck3 depot13)
   :method just-move-it
   (((not (traincar truck3)) :goal (0 6))
    ((move truck3 depot1 depot13 route66) :primitive (2 3))
    ((at truck3 depot1) :goal (0 2))
    ((connects route66 depot1 depot13 49) :goal (0 6))
    ((can-travel truck3 route66) :goal (0 6))))))
```

(The method names are references to the transplan domain; see `transplan/methods.pddl`.)

Although this looks complicated, it should be easy for a hierarchical planner to produce it as output. We will supply, in the near future, a solution checker that can take a structure of this kind and verify that it solves a problem.

For Phase 1 of the competition, we will ignore package volume, plus fuel, time, and water constraints. That is, `fuel-rate`, `fuel-waste`, and `water-rate` are set to 0, all packages have volume 0, and total elapsed time is not counted in the score of a planner.

In Phase 2 of the competition, these rates will become nontrivial. Then it will be possible for vehicles to fail to move for lack of fuel. Vehicles can refuel at depots (using the action `(liquid-transfer` *depot* `storage-tank` v `gas-tank` x`))`, but they have to have enough fuel to reach the depots.

In Phase 3 of the competition, new plans will be added to the the standard plan library, and planners will have to cope with them.

Ideally, planners should take the PDDL domain definition as input, suitably augmented with explicit advice, as explained in the PDDL Manual. Points will be taken off for the use of advice, although the exact formula has not been arrived at yet.

Some contestants may find it burdensome to have to retarget their planners to handle PDDL input. In that case, they are allowed to rewrite the entire domain spec in their own language, subject to the following provisos:

1. The planner must accept problems expressed in the `(define (problem ...))` notation.

2. The planner must output solutions in the form that is checkable by our forthcoming automatic checker.

3. The competition committee will decide how many advice points to take off for idiosyncratic notations. We won't penalize special notations per se; if they appear to be as neutral as the PDDL definition, they might lose zero points.

4. They won't be able to enter Phase 3 of the competition.

Further details will be made available later.

References

Scott Andrews, Brian Kettler, Kutluhan Erol, and James Hendler 1995 UM Translog: a planning domain for the development and benchmarking of planning systems. University of Maryland Technical Report CS-TR-3487.