Försättsblad till skriftlig tentamen vid Linköpings Universitet

Datum för tentamen	2012-08-16
Sal (1) Om tentan går i flera salar ska du bifoga ett försättsblad till varje sal och ringa in vilken sal som avses	T1
Tid	14-18
Kurskod	TDDD48
Provkod	TEN1
Kursnamn/benämning **Provnamn/benämning**	Automatisk planering Skriftlig tentamen
Institution	IDA
Antal uppgifter som ingår i tentamen	4
Jour/Kursansvarig Ange vem som besöker salen	Jonas Kvarnström,
Telefon under skrivtiden	ankn 2305 eller 0704-737579
Besöker salen ca kl.	ja
Kursadministratör/kontaktperson (namn + tfnr + mailaddress)	Anna Grabska Eklund, 2362, anna.grabska.eklund2liu.se
Tillåtna hjälpmedel	inga
Övrigt	
Vilken typ av papper ska användas, rutigt eller linjerat	valfritt
Antal exemplar i påsen	

Exam: TDDD48 Automated Planning 2012-08-16

Important Notes

Read the following before you begin!

- Though the questions are in English, you may **answer in Swedish** if you prefer!
- *Clear and comprehensible* explanations and motivations are always required. This does not necessarily mean that each answer should be a long essay. What is important is that all the relevant facts are present and clearly explained.
- Concrete examples or counterexamples may be useful as part of a motivation. If so, please make sure you include all relevant information about the example.

1 Properties of Classical Plans

The following questions relate to the general structure of classical plans and some general properties of such plans. We know the following:

- A (sequential) solution plan π for a classical problem instance P is *redundant* iff it is possible to remove one or more actions from π in such a way that the remaining actions, *in their original order*, still form a solution that achieves the goal. For example, the solution $[a_1, a_2, a_3, a_4, a_5, a_6]$ is redundant if $[a_1, a_2, a_3, a_6]$ or some other subsequence is also a solution.
- A solution plan π for a given problem instance is *minimal* iff there is no other solution for P that contains strictly fewer actions.

The standard **logistics domain**, where a set of packages should be transported to their destinations, is defined in Appendix A. You should do the following (see also the hint below):

a) Create a classical planning problem instance for this domain, including a clearly specified initial state and goal. Show a *redundant* solution for this problem instance. Show why the solution is redundant – in other words, indicate which actions can be removed from the solution. **(1 point)**

b) Create a classical planning problem instance for this domain, including a clearly specified initial state and goal. Show a solution for this problem instance that is *not* redundant but still not minimal. Explain / motivate clearly why the solution is not redundant and why it is not minimal. **(1 point)**

Clarifications and hints:

- Don't add too many objects or goals – you can generate redundant plans for quite small problem instances, and using large instances will take you more time than necessary.

2 Search Guidance

Given the size of a typical search space, a planner almost always needs some form of search guidance as opposed to doing blind search. Often such guidance takes the shape of a *heuristic function* that evaluates the "quality" of a certain search node, corresponding to one specific choice that can be made at a particular point in the search space.

a) Heuristic functions can yield *plateaus* in the search space. What is a plateau? Visualize one using part of a search space: Show a set of search nodes, a number of possible transitions between nodes, and the type of heuristic values that characterize a plateau. Show which nodes are part of the plateau and explain in words *why* they form a plateau. **(2 points)**

b) Given an optimization problem, plateaus can in some cases be handled simply by terminating search as soon as a plateau is found. Explain why this approach is generally not useful in classical planning. Also, explain at least one method that can be used in planning for dealing with ("escaping") plateaus. **(2 points)**

c) Is the h_{add} heuristic, also called h_0, admissible? If it is, motivate clearly why. If it is not, demonstrate using a counterexample: Show a state and goal for which h_{add} is *not* admissible, using the logistics planning domain defined in Appendix A. **(1 point)**

One possible alternative to a heuristic is the use of temporal control rules.

d) Write a useful control rule for the logistics domain as specified in Appendix A. The control rule must be specified formally, using the temporal modal operators *always* ($\Box$), *eventually* ($\Diamond$), *next* ($\circ$), and *until* (U) together with the modal goal operator, goal.

   Though the control rule does not have to use all of these modal operators, it must use at least two temporal operators as well as refer to the goal using the goal operator. **(2 points)**

3 Partial-Order Planning

We now consider partial-order planning using the **elevator domain** defined in Appendix B. For simplicity this domain only has a single lift, but there are still opportunities for partial ordering during the actual planning process! You should use the following problem instance:

```
(define (problem mixed-f6-p3-u0-v0-g0-a0-n0-A0-B0-N0-F0-r0)
    (:domain elevator)
    (:objects p0 p1 p2 - passenger
              f0 f1 f2 f3 f4 f5 - floor)
    (:init
        (above f0 f1) (above f0 f2) (above f0 f3) (above f0 f4) (above f0 f5)
        (above f1 f2) (above f1 f3) (above f1 f4) (above f1 f5)
        (above f2 f3) (above f2 f4) (above f2 f5)
        (above f3 f4) (above f3 f5)
        (above f4 f5)
        (origin p0 f0) (destin p0 f4)
        (origin p1 f3) (destin p1 f1)
        (origin p2 f5) (destin p2 f1)
        (lift-at f0)
    )
    (:goal (and (served p0) (served p1) (served p2)))
)
```

You should do the following (also make sure you read the notes below):

a) Show the *initial partial plan* π_0 generated for this problem instance by a typical partial-order planner, such as the PSP planner in the course book (corresponding to what we discussed during the lectures). This is the partial plan that corresponds to the first node ("root node") generated in the search space. Make sure that all relevant aspects of the actions involved are clearly indicated in the plan. **(1 point)**

b) Show *all* the immediate successors of the initial partial plan in the partial-order search space. That is, demonstrate all the different ways in which a standard PSP-like partially ordered planner might modify π_0 in a single step. If this should result in several almost identical partial plans, you may save time by showing a set of representative example partial plans and explaining (very clearly!) in text which additional successors exist. **(2 points)**

c) Show a complete solution plan for this problem instance. Make sure that actions are not temporally constrained relative to each other unless this is necessary in order to achieve the goal. **(1 point)**

Note: For each action in a partially ordered plan, you must clearly show each precondition *above* the action and each *effect* below the action. You must also indicate all other relevant structural features in the plan: precedence constraints (solid arrows), causal links (dashed arrows) and threats.

Note: As you see, there are quite a lot of true instances of `above()`. To reduce space requirements, you may illustrate all true instances of `above()` simply as "`above(...)`" in the same place where you would normally have written out all instances explicitly.

4 Hierarchical Task Networks

It is now time to define an HTN formulation of the elevator domain for a **TFD-like** procedure (Total-order Forward Decomposition) with totally ordered methods.

The "goal" will be represented through an initial task network consisting only of the non-primitive task transport-everyone(). The intention is for this task to ensure that everyone who is not yet served will be served (transported to his/her destination).

Your job is to specify one or more methods for the transport-everyone() task, as well as any other *non-primitive* tasks and methods that you want to use, to ensure that the task can eventually be decomposed into *primitive* tasks. The operators defined in Appendix B should be used as primitive tasks.

You may assume that each type is associated with a **type predicate** of the same name. For example, the type floor is accompanied by a type predicate floor(x).

For this question, you may choose a level of difficulty as follows:

- For **2 points**, define an HTN domain where each person is transported separately to his or her destination, without taking into account the fact that more than one person may have the same origin or destination. This may result in "inefficient" but still correct solutions.
- For **4 points**, define an HTN domain where you ensure that whenever the elevator stops at a certain floor to pick up someone, all persons waiting for an elevator at that floor will board, and whenever it stops in order for someone to depart, all persons currently on board and having this destination will depart.

Clarifications:

- For every method you create, you must specify which task it corresponds to, which preconditions it has, and which partially ordered set of parameterized subtasks it is decomposed into.
- JSHOP2 uses a somewhat different structure for tasks and methods compared to the book. You may choose to use either the JSHOP2 structure or the book structure, as long as it is indicated clearly which one you use.
- As always, exact syntax (such as where parentheses are placed) is not irrelevant but is less important than showing that you have understood the concepts involved. Please add explanations to clarify what the different parts of your definitions mean, if you are uncertain whether you follow the correct syntax precisely.

A The Logistics Domain

The standard **logistics domain** contains a set of *packages* that should be transported to their destinations. A package can be transported by *truck* between any two *locations* in the same city, and by *airplane* between special *airport* locations in different cities. Since trucks cannot deliver packages directly to other cities, and airplanes cannot visit arbitrary locations, delivering a package might require using a truck to move it to an airport, using an airplane to move it to another city and then once again using a truck to get the package to its final destination.

We assume the following types of objects:

- thing, with subtypes package and vehicle
- vehicle, with subtypes truck and airplane
- location, with subtype airport

A model of this domain may include the following **operators**.

- load-truck(package, truck, location) loads a package into a truck, given that they are both at the same location. The truck can hold an arbitrary number of packages.
- unload-truck(package, truck, location) unloads a package from a truck in the current location.
- load-plane(package, airplane, location) loads a package into a plane, given that they are both at the same location. The plane can hold an arbitrary number of packages.
- unload-plane(package, airplane, location) unloads a package from an airplane in the current location.
- drive-truck(truck, location, location) drives a truck between locations in the same city.
- fly-plane(plane, airport, airport) flies a plane between two airports in different cities.

We assume the following **predicates** are available:

- at(thing,loc) – the package or vehicle thing is at the location loc. Note that a package "at" a certain location cannot be "in" a vehicle in the same state.
- in(x,vehicle) – the package x is in the given vehicle.
- same-city(loc1,loc2) – the given locations are in the same city. Note that any location is in the same city as itself.

We assume a **typed** domain model where each operator parameter and predicate parameter is given a specific type and cannot take on values outside that type. For example, if t_1 is a truck, an expression such as at(t_1, t_1) is not merely false but syntactically incorrect. Should you need **type predicates** at some point, you can assume that thing(x), package(x), vehicle(x), truck(x), airplane(x), location(x) and airport(x) are inferred automatically by the planner given the typed values you define in the problem specification.

We can then define the operators more formally as shown on the following page:

- load-truck(package *pkg*, truck *trk*, location *loc*)
  Precondition: at(*pkg*, *loc*) ∧ at(*trk*, *loc*)
  Effects: ¬at(*pkg*, *loc*), in(*pkg*, *trk*)
- load-airplane(package *pkg*, airplane *plane*, location *loc*)
  Precondition: at(*pkg*, *loc*) ∧ at(*plane*, *loc*)
  Effects: ¬at(*pkg*, *loc*), in(*pkg*, *plane*)
- unload-truck(package *pkg*, truck *trk*, location *loc*)
  Precondition: in(*pkg*, *trk*) ∧ at(*trk*, *loc*)
  Effects: ¬in(*pkg*, *trk*), at(*pkg*, *loc*)
- unload-airplane(package *pkg*, airplane *plane*, location *loc*)
  Precondition: in(*pkg*, *plane*) ∧ at(*plane*, *loc*)
  Effects: ¬in(*pkg*, *plane*), at(*pkg*, *loc*)
- drive-truck(truck *trk*, location *from*, location *to*)
  Precondition: at(*trk*, *from*) ∧ same-city(*from*, *to*)
  Effects: ¬at(*trk*, *from*), at(*trk*, *to*)
- fly-airplane(airplane *plane*, airport *from*, airport *to*)
  Precondition: at(*plane*, *from*)
  Effects: ¬at(*plane*, *from*), at(*plane*, *to*)

B The Elevator Domain

The following is a variation of the standard *elevator domain*, which contains a set of *passengers* that should be transported to their destination *floors*. For simplicity, we only use a single elevator/lift. We assume a **typed** domain model where each operator parameter and predicate parameter is given a specific type and cannot take on values outside that type.

Each passenger is represented by a distinct object (p1, p2, p3, ...). Since classical planners generally do not support numeric values, floors are also represented as distinct objects (f1, f2, f3, ...) together with an above predicate, rather than simply using numbers and ">" (greater than).

Note that there is no predicate directly specifying the current floor of a passenger. Instead, there are fixed (constant) predicates specifying the origin and destin(ation) of the passenger, together with two other predicates that are modified by operators: boarded and served. If the passenger has not boarded and is not served, he/she must be at the origin. If the passenger has boarded but is not served, he/she is in the lift. If the passenger is not currently boarded and is served, he/she is at the destination. This is essentially a clever way of encoding control knowledge in the fundamental structure of the domain without the use of explicit temporal control formulas, with preconditions ensuring one cannot board or debark an elevator unnecessarily.

```
(define (domain elevator)
  (:requirements :strips)
  (:types object
          passenger - object  ;; Passengers p1, p2, ...
          floor - object)     ;; Floors f1, f2, ...

(:predicates
    (origin ?person - passenger ?floor - floor)
    ;; initially, ?person is at ?floor
    ;; (remains true even after the person has moved)

    (destin ?person - passenger ?floor - floor)
    ;; the destination of ?person is ?floor

    (above ?floor1 - floor  ?floor2 - floor)
    ;; ?floor2 is located above ?floor1
    ;; (for example, f4 may be above f3, f2 and f1)

    (boarded ?person - passenger)
    ;; true if ?person is on board the lift

    (served ?person - passenger)
    ;; true if ?person has arrived at her destination

    (lift-at ?floor - floor)
    ;; current position of the lift is at ?floor
)
```

Actions are shown on the following page.

```
(:action board
  :parameters (?f - floor ?p - passenger)
  :precondition (and (lift-at ?f) (origin ?p ?f))
  :effect (boarded ?p))

(:action depart
  :parameters (?f - floor ?p - passenger)
  :precondition (and (lift-at ?f) (destin ?p ?f) (boarded ?p))
  :effect (and (not (boarded ?p)) (served ?p)))

(:action up ;; Moves the lift up
  :parameters (?f1 - floor ?f2 - floor)
  :precondition (and (lift-at ?f1) (above ?f1 ?f2))
  :effect (and (lift-at ?f2) (not (lift-at ?f1))))

(:action down ;; Moves the lift down
  :parameters (?f1 - floor ?f2 - floor)
  :precondition (and (lift-at ?f1) (above ?f2 ?f1))
  :effect (and (lift-at ?f2) (not (lift-at ?f1))))
)
```