Information page for written examinations at Linköping University

Examination date	2011-06-01
Room (1) If the exam is given in different rooms you have to attach an information paper for each room and mark intended place	U14
Time	8-12
Course code	TDDD48
Exam code	TEN1
Course name **Exam name**	Automatisk planering Skriftlig tentamen
Department	IDA
Number of questions in the examination	5
Teacher responsible/contact person during the exam time	Jonas Kvarnström
Contact number during the exam time	0704-737579
Visit to the examination room approx.	09.00
Name and contact details to the course administrator (name + phone nr + mail)	Anna Grabska Eklund, 2362, anna.grabska.eklund@liu.se
Equipment permitted	inga
Other important information	
Which type of paper should be used, cross-ruled	valfritt

Exam: Automated Planning 2011-06-01

Please note that you may **answer in Swedish** if you prefer!

1 Heuristics in Classical State Space Planning

Many classical planners build on forward-chaining search together with heuristics guiding search towards promising areas of the state space. Often these heuristics build on the use of *delete relaxation*, where delete effects (negative effects) are ignored, resulting in a simpler problem that can be solved in order to estimate the difficulty of the true planning problem.

a) We have discussed the h_m family ($m \geq 1$) of domain-independent admissible heuristics. These heuristics build on delete relaxation, but add further simplifications.

   Explain the meaning of $h_m(s)$, where s is a state. That is, what property of the state and the planning problem does $h_m(s)$ calculate for different values of m? You do not have to specify this as a formula / equation, but should provide a clear explanation. **(2 points)**

b) The h_0 (h-zero) heuristic has one fundamental difference from the standard h_m family of heuristics. What is the difference? In what way does this affect the heuristic (positively and/or negatively) in terms of strength as well as applicability? **(2 points)**

2 Neo-Classical Planning

Recall that planning techniques that generate classical plans for classical planning domains, but that use alternative and non-trivial representations of search spaces, are called *neo-classical*. Such techniques include SAT planning (planning based on translations into propositional satisfiability) and GraphPlan (based on planning graphs).

a) In SAT planning, we *may* use *complete exclusion axioms*. What is the purpose of such axioms and when would we want to avoid using them? **(1 point)**

b) In SAT planning, we need *frame axioms* (for example, *explanatory* frame axioms) to ensure that the SAT solver gives correct results that correspond to valid plans. Explain what could go wrong, and how, if frame axioms were not included in the SAT translation of a planning problem. **(1 point)**

c) GraphPlan: Planning based on planning graphs iterates over two phases: Forward graph expansion, where a new graph level is added, and backward graph search.

   Doing forward graph expansion to level n is more efficient than completely exploring the state space to a search depth of n using plain forward-chaining search with a depth limit. Why is this the case, and why can forward graph expansion "get away with" doing less work than plain forward search? **(2 points)**

d) GraphPlan: Why is backward search in a planning graph more efficient than simply doing standard backward search in the standard state space? **(1 point)**

e) GraphPlan: Each level in a planning graph (except level zero) contains two different layers. Explain for each layer which kind of "entities" it contains and when these entities are considered *mutually exclusive*. **(1 point)**

3 Partial-Order Planning

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

```
(define (problem mixed-f6-p3-u0-v0-g0-a0-n0-A0-B0-N0-F0-r0)
    (:domain miconic)
    (: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 f3) (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.

   For this particular task you do not have to illustrate each successor plan graphically, in case this would lead to excessive work. Instead, you can explain clearly in writing how each successor would extend π_0. However, you still need to indicate *all* changes that would be made for each successor, including new constraints, relations and links! **(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 **PFD-like** planner (Partial-order Forward Decomposition): Subtasks can be partially ordered in a method specification, and plans can interleave subtasks originating in different parent tasks (you do parts of one task, then parts of another, then continue with the first task).

We represent the "goal" through an initial task network consisting of one or more *unordered* non-primitive tasks of the form transport(x, from, to), where x is a person who should be transported from floor from to floor to.

You should specify the required set of *non-primitive* tasks and methods for this domain, using the operators defined in Appendix A as *primitive* tasks. That is, you should specify non-primitive tasks and methods that decompose the initial set of tasks into subtasks ensuring that the objective is achieved, in such a way that you eventually reach the predefined primitive tasks. You may assume that each type is associated with a **type predicate** of the same name. For example, the type person is accompanied by a type predicate person(x).

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

- For **1 point**, 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.
- For **3 points**, define an HTN domain where you ensure that whenever the elevator stops at a certain floor to pick up someone, all persons waiting at that floor may depart, and whenever it stops in order for someone to depart, all persons currently on board with this destination may depart.

  Note that you cannot do this simply by iterating or recursing over all persons at the current floor or all persons in the elevator, since this does not work well with the initial task network containing a number of separate "transport" tasks. Instead, it must be done by allowing the planner to interleave tasks where necessary.

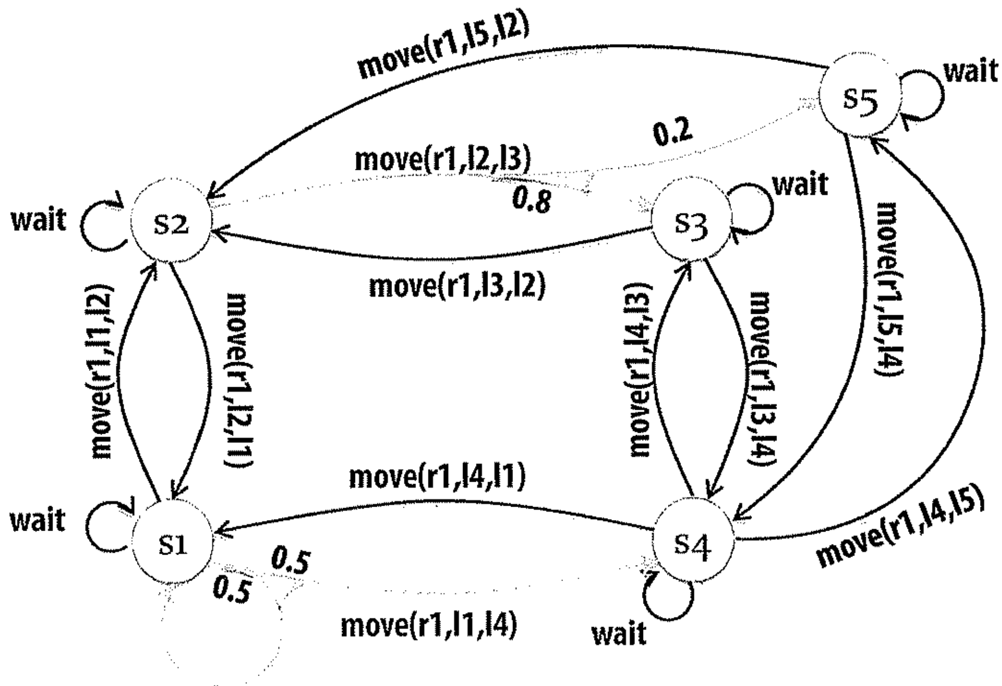
  Note that for the purposes of this exam, it is sufficient to leave the HTN planner enough freedom so that it *can* choose to generate interleaved plans. You do not have to *force* the planner to interleave the plans in the best way.
- For **5 points**, define an HTN domain as for 3 points, but also ensure that the elevator can never skip a floor where there are people to pick up or "deliver" (assuming the elevator is of infinite capacity). The elevator must also never stop at a floor where there is no one to pick up or deliver, so simply stopping at *all* floors is not a solution.

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 of these structures, as long as it is indicated clearly which one you use.
- Note that if there is an initial task transport(x, from, to) where x is not initially at from, planning must naturally fail!
- As always, exact syntax (such as where parentheses are placed) is less important than showing that you have understood the concepts involved. Feel free to add explanations to clarify what the different parts of your definitions mean, if you are uncertain whether you follow the correct syntax precisely.

5 Planning with Markov Decision Processes

The following example of a stochastic process has been used during the lectures:



Assume that the cost of each move action is 100, while the cost of waiting is unspecified. Rewards for visiting states are also unspecified. Assume a discount factor of 0.9.

a) Suppose that our objective is to visit states s2 and s4 *repeatedly* (over and over again). For example, we might want to see regularly what happens at each of those two locations.

   Specify costs (for each wait action) and rewards (for all states) that ensure that this will happen, regardless of which state we start in. **(1 point)**

b) Begin with an initial policy π_0 such that for any state s, $\pi_0(s) = \text{wait}$. Then perform two full steps of *policy iteration* given your own costs and rewards, creating the policies π_1 and π_2. **(2 points)**

A The Elevator Domain

The *elevator domain* contains a set of *passengers* that should be transported to their destination *floors*. For simplicity, we use a single lift in this simplified domain formulation. 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.

```
(define (domain miconic)
  (:requirements :strips)
  (:types object
          passenger - object
          floor - object)

(:predicates
    (origin ?person - passenger ?floor - floor)
    ;; entry of ?person is ?floor

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

    (above ?floor1 - floor  ?floor2 - floor)
    ;; ?floor2 is located above ?floor1

    (boarded ?person - passenger)
    ;; true if ?person has boarded 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
)

(: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))))
)
```