Försättsblad till skriftlig tentamen vid Linköpings Universitet

(fylls i av ansvarig)

Datum för tentamen	*2010-08-16*
Sal	*U3*
Tid	*8-12*
Kurskod	*TDDD48*
Provkod	*TEN1*
Kursnamn/benämning	*Automatisk planering*
Institution	*IDA*
Antal uppgifter som ingår i tentamen	*4*
Antal sidor på tentamen (inkl. försättsbladet)	*7*
Jour/Kursansvarig	*Jonas Kvarnström*
Telefon under skrivtid	*0704-737579*
Besöker salen ca kl.	*Kl.09:00*
Kursadministratör (namn + tfnnr + mailadress)	*Anna Grabska Eklund* *Ankn. 23 62, annek@ida.liu.se*
Tillåtna hjälpmedel	*inga*
Övrigt (exempel när resultat kan ses på webben, betygsgränser, visning, övriga salar tentan går i m.m.)	
Vilken typ av papper ska användas, rutigt eller linjerat	*valfritt*
Antal exemplar i påsen	

Exam: Automated Planning 2010-08-16

Please note that you may answer in Swedish.

1 Classical Plans (4 points)

Recall that in standard classical planning problems, no function symbols are allowed, goals constrain only the final state reached by a plan, and preconditions and effects consist of simple sets of positive and negative literals, without support for disjunction, quantification or conditional effects.

Let $P_1 = (O, s_0, g_1)$ and $P_2 = (O, s_0, g_2)$ be two classical planning problems that share the same operators and initial state. Let $\pi_1 = [a_1, \ldots, a_n]$ be a solution of length n for P_1, and let $\pi_2 = [b_1, \ldots, b_n]$ be a solution of the same length for P_2.

a) Suppose that the concatenated action sequence $\pi_1 \cdot \pi_2 = [a_1, \ldots, a_n, b_1, \ldots, b_n]$ happens to be *executable* starting at state s_0. Can we then be certain that it is also a *solution* for P_2? Motivate clearly why this is or is not the case. **(1 point)**

b) Suppose that no operator in O has negative effects. Can we then be certain that the interleaved action sequence $[a_1, b_1, a_2, b_2, \ldots, a_n, b_n]$ is a solution for $(O, s_0, g_1 \cup g_2)$? Motivate clearly why this is or is not the case. **(1 point)**

c) Let $\Pi = \{\pi \mid \pi \text{ is applicable to } s_0\}$ be the set of all action sequences that are executable starting at s_0. Is Π *always* finite, *sometimes* finite or *never* finite? Explain why. **(1 point)**

d) Let $S = \{\gamma(s_0, \pi) \mid \pi \text{ is applicable to } s_0\}$ be the set of all states that are reachable through executable action sequences starting at s_0. Is S *always* finite, *sometimes* finite or *never* finite? Explain why. **(1 point)**

Note that points are given mainly for clear and understandable explanations and motivations.

2 Planning Graphs (6 points)

We will now take a closer look at **planning graphs** using the standard **logistics domain**, where a set of packages should be transported to their destinations. This domain is defined in Appendix A. We will use a simple problem instance containing the packages $\{p_1, p_2\}$, the trucks $\{t_1, t_2\}$ and the locations $\{l_1, l_2\}$, none of which are airports.

In the initial state, we know that $\mathsf{at}(p_1, l_1)$, $\mathsf{at}(p_2, l_2)$, $\mathsf{at}(t_1, l_2)$, and $\mathsf{at}(t_2, l_2)$. All locations are in the same city: $\{\mathsf{same\text{-}city}(l, l') \mid l, l' \in \{l_1, l_2\}\}$. No packages are loaded into vehicles.

The goal is that $\mathsf{at}(p_1, l_1)$, $\mathsf{at}(p_2, l_1)$, $\mathsf{at}(t_1, l_2)$, and $\mathsf{at}(t_2, l_2)$.

You may assume a typed representation, where "nonsense" propositions such as $\mathsf{at}(t_1, t_2)$ and actions such as $\mathsf{drive\text{-}truck}(p_1, p_2, p_1)$ cannot occur. Type predicates such as $\mathsf{truck}(t)$ are not used by the operators defined in the appendix and therefore do not have to be present in the planning graph. The same-city predicate never changes and can therefore also be omitted from the graph.

You should do the following:

a) Generate a planning graph for the problem instance defined above, containing a total of three proposition levels. That is, you can stop expanding the graph after reaching three proposition levels, without having to continue until a fixpoint is reached.

   You must explicitly show all information that is normally part of a planning graph, including for example mutual exclusion (mutex) relations, though some information may be easier to provide in the shape of text or a table after the actual graph. Make sure to leave sufficient room for all required arrows! **(3 points)**

b) Explain when entities in a planning graph are mutually exclusive. Also explain how mutual exclusion at one level affects the next level of the graph and how this in turn affects the size of the complete planning graph. Give a concrete example from the planning graph you created above, showing one way in which the graph would have been different if mutual exclusion had not been considered. **(1 point)**

c) Explain how planners such as GraphPlan benefit from the use of a planning graph to improve performance compared to more straight-forward methods such as plain forward- or backward-chaining. How is the planning graph used during search and how does it help the planner find a plan more efficiently? Note that we are *not* referring to the fact that planning graphs can be used "indirectly" to define heuristic functions for standard heuristic forward-chaining search! **(2 points)**

3 Partial-Order Planning (6 points)

A similar but slightly different problem instance will now be considered in the context of partial-order planning. We once again use the standard **logistics domain** as defined in Appendix A, but now use a problem instance containing the packages $\{p_1, p_2\}$, the trucks $\{t_1, t_2\}$ and the locations $\{l_1, l_2, l_3\}$, none of which are airports.

In the initial state, we know that $\mathsf{at}(p_1, l_1)$, $\mathsf{at}(p_2, l_2)$, $\mathsf{at}(t_1, l_2)$, and $\mathsf{at}(t_2, l_2)$. All locations are in the same city: $\{\mathsf{same\text{-}city}(l, l') \mid l, l' \in \{l_1, l_2, l_3\}\}$. No packages are loaded into vehicles.

The goal is that $\mathsf{at}(p_1, l_3)$, $\mathsf{at}(p_2, l_3)$, $\mathsf{at}(t_1, l_2)$, and $\mathsf{at}(t_2, l_2)$.

You should do the following:

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. This is the partial plan that corresponds to the first node generated in the search space. **(1 point)**

b) Show all the immediate successors of the initial partial plan. That is, demonstrate all the different ways in which a standard PSP-like partially ordered planner might extend π_0 in a single step.

   For this particular task you do not have to illustrate each successor plan graphically. 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, including new constraints and relations! **(2 points)**

c) Show a partial plan for this problem in which one or more threats appear. Indicate all threats clearly. (Note that you might be able to extend this plan into a solution below.) **(2 points)**

d) Show a complete solution plan for this problem instance, making good use of both trucks to efficiently resolve all goals. 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.

4 Hierarchical Task Networks (4p)

We continue by defining an HTN (hierarchical task network) formulation of the logistics domain under the assumption that a **PFD-like** procedure is used (**Partial-order** Forward Decomposition): Methods are partially ordered and subtasks originating in different parent tasks can be interleaved (you do parts of one task, then parts of another, then continue with the first task).

A logistics goal is then represented through an initial task network consisting of one or more non-primitive tasks of the form deliver(x, from, to), where x is a package which is initially at from and should be delivered at to. For example, there may be nodes for the tasks {deliver(package1,loc1,loc2), deliver(package2,loc1,loc3), deliver(package3,loc3,loc5)}.

You should:

- Specify the required set of *non-primitive* tasks and methods for the logistics domain, using the six operators defined in Appendix A as primitive tasks. In other words, you should specify non-primitive tasks and methods that decompose an initial set of tasks on the form deliver(x, from, to) into subtasks ensuring that package x is eventually delivered at to, in such a way that you eventually reach the predefined primitive tasks. Note that if x is not initially at from, the package cannot be delivered from that location and the method you defined must naturally fail!

  Your methods must be general in the sense that they can be applied to arbitrary problem instances. Solution plans must not include actions (primitive tasks) that "do nothing", such as driving from a location to the same location.

  For **2 points**, you are allowed to create inefficient solution plans in the specific sense that trucks and airplanes only move a single package at a time.

  For **4 points**, you must make sure your methods *can* result in plans that are optimal in terms of the number of actions used.

  This merits some further discussion. HTN planners can often make choices, random or informed, as to which method should be used in order to accomplish a specific task. For example, it may choose to deliver five packages using the same truck, or split them in various ways between two different trucks. Determining in advance which variation leads to an optimal plan can be very difficult. Therefore, we only require that you leave sufficient freedom so that the optimal plan *can* be found if the PFD procedure just happens to make the best possible non-deterministic choices. For example, for 4 points your methods must allow an arbitrary number of packages to be moved by a truck at the same time by interleaving subtasks.

Clarifications:

- For every new 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.
- 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.

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**. For Simple Task Network planning, these operators correspond directly to **primitive tasks**.

- 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 a location must be in the same location 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 $\mathsf{at}(t_1, t_1)$ is not merely false but incorrect. Nevertheless, we provide the following **type predicates** that may be useful in some situations: thing(x), package(x), vehicle(x), truck(x), airplane(x), location(x) and airport(x).

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