Försättsblad till skriftlig tentamen vid Linköpings Universitet

Datum för tentamen	2013-10-24
Sal (1) Om tentan går i flera salar ska du bifoga ett försättsblad till varje sal och <u>ringa in</u> vilken sal som avses	KÅRA
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.	
Kursadministratör/kontaktperson (namn + tfnr + mailaddress)	Anna Grabska Eklund, ankn. 2362, anna.grabska.eklund@liu.se
Tillåtna hjälpmedel	inga
Övrigt	
Vilken typ av papper ska användas, rutigt eller linjerat	Valfritt
Antal exemplar i påsen	

Exam: TDDD48 Automated Planning 2013-10-24

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 you have chosen to use. What is relevant naturally depends on how you use the example.

1 Classical Planning

Recall that in standard classical planning problems, no function symbols are allowed, goals constrain only the final state reached by a plan, and preconditions, effects and goals can be represented as simple sets (conjunctions) of positive and negative literals, without 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 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)**

b) 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* infinite, *sometimes* infinite or *never* infinite? Explain why. **(1 point)**

c) 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* infinite, *sometimes* infinite or *never* infinite? Explain why. **(1 point)**

Note that clear and comprehensible explanations and motivations are required. In some cases, examples or counterexamples may be useful as part of a motivation.

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

The goal is that at(p_1, l_1), at(p_2, l_1), at(t_1, l_2), and at(t_2, l_2).

You may assume a typed representation, where "nonsense" propositions such as at(t_1, t_2) and actions such as drive-truck(p_1, p_2, p_1) cannot occur. Type *predicates* such as truck(t) are not used by the operators defined in the appendix and therefore do not have to be present in the planning graph. You may also omit the same-city predicate from the graph, as it may otherwise be too complex to keep track of the graph "manually".

You should do the following:

a) Generate a planning graph for the problem instance defined above, containing a total of three proposition levels including the initial level. 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. This information may be easier to provide as a table after the actual graph!

   We strongly recommend that you make a quick sketch of the planning graph on a separate paper before constructing a final version, since it can be difficult to determine in advance how large the graph will become. When you "clean it up", make sure to leave sufficient room for all required arrows. You may introduce abbreviations for actions and propositions as long as they are clear and unambiguous. **(2 points)**

b) Explain when entities in a planning graph are mutually exclusive. Also explain how mutual exclusion at one level affects which entities are present in 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 specific way in which the graph would have been different if mutual exclusion had not been considered. **(1 point)**

c) Above, you have shown how a planning graph can be created and expanded. You should now explain how planners such as GraphPlan actually use this graph during planning. For example, assuming that a graph with n layers has been created, how does GraphPlan use this graph to try to find a plan? When does GraphPlan terminate (give up)? **(1 point)**

3 Partial-Order Planning

We now consider partial-order planning. We use the standard **logistics domain** as defined in Appendix A, with a problem instance containing the packages $\{p_1, p_2\}$, the trucks $\{t_1, t_2\}$ and the *three* locations $\{l_1, l_2, l_3\}$, none of which are airports.

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

The goal is that at(p_1, l_3), at(p_2, l_3), at(t_1, l_2), and 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 ("root 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! **(1 point)**

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.) **(1 point)**

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

We continue by defining an HTN (hierarchical task network) formulation of the full logistics domain, where packages may be in different cities and both airplanes and trucks may have to be used. We assume a **TFD-like** procedure is used (Total-order Forward Decomposition), with totally ordered methods. The planner therefore cannot automatically interleave subtasks originating in different parent tasks.

The six operators defined for this domain in Appendix A can be used as primitive tasks and do not have to be specified by you. The predicates used in the appendix will also be available for use in your HTN formulation, as well as this additional predicate:

- dest(package,loc) – true iff the given package should be delivered to the location loc. This predicate is *fixed* in the sense that it never changes: It does not become false even if the package is already at its destination.

A logistics goal is then represented through the initial specification of the dest predicate together with an initial task network consisting of a single non-primitive task deliver-all(), corresponding to the task of delivering all packages that are not yet at their destinations.

You should:

- Specify one or more methods decomposing the non-primitive task deliver-all() described above into subtasks.
- For any non-primitive subtasks you invent, specify one or more methods decomposing those into subtasks as well, so that any decomposition of deliver-all() eventually reaches the pre-defined primitive tasks.

For full points **(3 points)**, your solution must satisfy the following:

- A truck must never go back to a location it has previously visited in order to load another package. (In this case, it would have been better to load the package the first time it visited that location.)
- If a truck visits a certain location, it must not leave while it is carrying a package having that location as destination.
- Solution plans must not include actions (primitive tasks) that "do nothing", such as driving from a location to the same location.

Clarifications and hints:

- For every new method you create, you must specify which task it corresponds to, which preconditions it has, and which sequence of parameterized subtasks it is decomposed into.
- JSHOP2, which we used for the labs, 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.

- Hint: How do you deliver a package if the destination is in the same city? How do you deliver it to a destination in another city? These are different ways of delivering the package.

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 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*)