Försättsblad till skriftlig tentamen vid Linköpings Universitet

Datum för tentamen	2011-08-18
Sal (1) Om tentan går i flera salar ska du bifoga ett försättsblad till varje sal och ringa in vilken sal som avses	TER4
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	0704-737579
Besöker salen ca kl.	15:00
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	
Antal exemplar i påsen	

Exam: Automated Planning 2011-08-18

Please note that you may **answer in Swedish** if you prefer!
Note that some pages have clarifications at the end!

1 Classical Planning

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 $[a_1, a_2, a_3, a_6]$ 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:

a) Create a classical planning problem instance for this domain and show a solution π for this problem instance, such that π is redundant (and therefore not minimal). Show why the solution is redundant – in other words, show which actions can be removed from the solution. **(1 point)**

b) Create a classical planning problem instance for this domain and show a solution π for this problem instance, such that π is *not* redundant but still not minimal. Explain / motivate clearly why the solution is not redundant and why it is not minimal. **(1 point)**

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.

c) 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 (achieves the goals of P_2)? Motivate clearly why this is or is not the case. If you want to provide an example, you may use the logistics domain or any other domain. **(1 point)**

Clarifications and hints:

- Problem instances must specify goals and *complete* initial states.
- 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 in Classical Planning

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 (including search nodes and transitions between nodes). **(1 point)**

b) The technique of *delete relaxation* is used as one important aspect of many different heuristic functions. Assume for this question that we *only* apply pure delete relaxation, with no additional relaxations. How does delete relaxation transform a planning problem / domain? Also, how is the transformed problem used to calculate a heuristic function given a state for the original problem? **(2 points)**

c) Temporal control rules can serve as an alternative or as a complement to heuristics in forward state space search. Give a conceptual description of temporal control rules. The following conceptual description of heuristic functions may serve as inspiration:

   "Each node in the search space can be seen as corresponding to a single specific world state. A heuristic function takes such a state and attempts to estimate the remaining distance from this state to the nearest goal state. A heuristic search algorithm can then use the heuristic function to determine, in one of many different ways, which among the currently expanded nodes should be explored first. This does not actually reduce the size of the search space, since even low-priority nodes remain to be searched later if necessary. However, with a good heuristic, an algorithm can begin by searching those parts of the search space where solutions are more likely to be found, leaving other parts to be searched later if the heuristic turned out to be misleading."

   You should *not* feel constrained to follow exactly this pattern – indeed, you may *need* to write differently in order to properly describe temporal control rules. However, the example should serve to illustrate the desired level of detail and abstraction. **(2 points)**

3 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).

For SAT planning, we have the following questions:

a) SAT-based planning transforms a planning problem into a satisfiability problem. Name one advantage of doing such a transformation rather than attacking the planning problem directly. **(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 property of the world is modeled by the frame axioms (what the frame axioms "mean"). Also explain with a simple example what could go wrong during planning if frame axioms were not included in the SAT translation of a planning problem. **(2 points)**

For planning with planning graphs, we will continue using the standard **logistics domain,** where a set of packages should be transported to their destinations. This domain is defined in Appendix A. Since large planning graphs tend to be difficult to handle manually, we will use a very simple problem instance containing the packages $\{p_1, p_2\}$, a single truck $\{t_1\}$ 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), and at(t_1, 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), and at(t_1, l_2).

You may assume a typed representation, where "nonsense" propositions such as at(t_1, p_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. The same-city predicate never changes and can therefore also be omitted from the graph.

You should do the following:

c) Generate a planning graph for the problem instance you defined, 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! It is probably a good idea to make a quick sketch on a separate paper before making the final version to be handed in. **(2 points)**

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 predefined 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*) ∧ 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*)