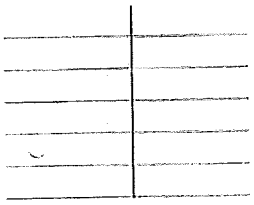
Försättsblad till skriftlig tentamen vid Linköpings universitet

Datum för tentamen	2018-08-23
Sal (1)	TER3(7)
Tid	8-12
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	013-28 23 05 eller 0704-737579
Besöker salen ca klockan	ca 09:00
Kursadministratör/kontaktperson (namn + tfnr + mailaddress)	Anna Grabska Eklund, ankn. 2362, anna.grabska.eklund@liu.se
Tillåtna hjälpmedel	inga
Övrigt	
Antal exemplar i påsen	

TDDD48 Automated Planning 2018-08-23

Important Instructions: Read before you begin!

- Though the questions are in English, feel free to answer in Swedish if you prefer! **Det går bra att skriva på svenska!**
- Please, write clearly (block letters if necessary), and use the *lined* side of the paper except for diagrams! Checked paper works well for math but tends to make text difficult to read...



- *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.
- To see if you have succeeded, turn the tables and **try to misunderstand.** Specifically, after you write an answer, go back and see if you can creatively misinterpret the answer. If so, *clarify, clarify, clarify*!
- A good way to show your knowledge is to write explanations that can be understood by someone *else* who *does not already know* the correct answer.

  If your answer explains a topic well enough that a fellow student could *learn* something new from the answer and/or apply it in practice, you have probably succeeded.

  Conversely, if we ask how HTNs work, saying that "there are methods and operators and you construct a tree using patterns and there can be constraints too, and there's no goal" does not provide much in terms of *useful* information, and certainly will not give you a full score.
- 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.
- When asked to provide examples or illustrate something through a planning problem instance, save time by *keeping the example as small as possible.*

Notation and Terminology

To avoid unnecessary confusion, we will hint at some of the terminology that you have hopefully learned during the course. **Please be careful to use the correct words!**

- **Predicate** (name/symbol) – at, on, raining
- **Object** (name/symbol) – A, B, red, truck1, helicopter8, location4
- **Variable** – *truck*, *location* – in italics
- **Atom** – at(mytruck, overthere), at(*truck*, location4), raining
- **Ground atom** – at(mytruck, overthere), raining
- **Fact** – typically a ground atom
- **Literal** – atom or negated atom
- **Ground literal**
- **Formula** – combination of atoms using connectives
- **State** – a full specification of the "configuration" of the world (to the extent it is modeled in the problem specification)
- **Goal formula, goal state, set of goal states, goal fact, ...**
- $h^*(s)$ – the cost of an optimal plan reaching a goal state from state s

1 General Concepts in Automated 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 consist of simple sets of positive and negative literals, without disjunction, quantification or conditional effects.

a) Let $P = (O, s_0, g)$ be a classical planning problem. Let $\Pi = \{\pi \mid \pi \text{ is applicable to } s_0\}$ be the set of all action sequences that are executable starting in the initial state s_0. Is Π *always* infinite, *sometimes* infinite or *never* infinite? Motivate and explain why. **(1 point)**

b) Continuing in the context of the previous question, let $S = \{\gamma(s_0, \pi) \mid \pi \in \Pi\}$ 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? Motivate and explain why.

(Here we have extended the state transition function $\gamma(s, a)$ to operate on sequences π in the natural way, by applying each action in π in the order of execution.) **(1 point)**

c) 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 $\pi_2 = [b_1, \ldots, b_m]$ be a solution of length m for P_2.

Suppose that the concatenated action sequence $\pi_1 \cdot \pi_2 = [a_1, \ldots, a_n, b_1, \ldots, b_m]$ 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)? Show clearly that this *is* or *is not* the case. If you want to provide an example or counterexample, try to keep it short and simple. **(2 points)**

d) Given a domain, problem instance and starting state, can you easily and quickly find the number of states reachable from the starting state? If so, show how. If not, motivate clearly. **(1 point)**

2 Partial-Order Planning

We now consider partial-order planning. We use the standard **logistics domain** as defined in Appendix A; please familiarize yourself with it before continuing.

We will use 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 $\mathsf{at}(p_1, l_1)$, $\mathsf{at}(p_2, l_2)$, $\mathsf{at}(t_1, l_2)$, 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 $\mathsf{at}(t_2, l_2)$.

Below you will be required to **demonstrate a number of partially ordered plans** that could be encountered during the planning process. Unless we explicitly say otherwise, you must clearly show the following in each such plan:

- For each action, all preconditions *above* the action and all effects *below* the action. (Or, if you write them horizontally: Preconditions to the left, effects to the right.)
- All other relevant structural features in the plan: precedence constraints (solid arrows), causal links (dashed arrows) and threats.

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 or the planning procedure illustrated during the course lectures. This is the partial plan that corresponds to the unique first node ("root node") generated in the search space. Think carefully about what the *very first* node is! **(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* additions that would be made relative to the initial partial plan, including new constraints and relations as discussed above! **(2 points)**

c) Show a partial plan for this problem in which one or more threats appear. Indicate all threats clearly, if there is more than one, and explain why they are threats. (Note that you might be able to extend this plan into a solution in the next subquestion.) **(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)**

3 Search Techniques

Note: A couple of questions are identical or similar to those on the previous exam. This is intended as a test to see to what extent students study and learn from participation.

Recall that the FF (FastForward) planner pioneered the use of *helpful actions*, a specially designated subset of the applicable actions in any particular search state. FF used these actions in a particular way together with enforced hill climbing. However, other planners have used them differently, through different ways of treating the *search (priority) queue*.

a) How are *dual queues* used to achieve a balanced use of helpful actions? For full points, you need to provide a clear description of the actual use of dual queues and how they are integrated in the search process, at a level close to pseudo-code. **(2 points)**

> **Hints** – note that *we are not interested in the answers to any of the hint questions*, you can simply use them to lead you towards the correct answers for the *actual* questions!
> - Hint: Dual queues can be used with many forms of best-first search, similar to A*. What does this tell you about how the search is structured?
> - Hint: What entities are usually present in a search queue/space? What does this tell you about how helpful actions may play a role?

b) What is the difference between dual queues and *boosted queues*? **(1 point)**

Lifted planning is another general technique applicable to a variety of search spaces and planning algorithms. During the lectures, we specifically discussed *lifted partial-order planning* as well as *lifted backward goal-space planning*, but since the technique itself is general, it could also be applied to for example HTN planning.

c) **Explain** what lifted planning is in general.

Then give a concrete example of why lifting can be useful when generating partial-order plans in particular. In this example you should:

- **Show** a small lifted partial-order plan structure (or a sufficiently complete and informative fragment of one),
- **Contrast** it against a non-lifted alternative structure that you also include in your answer, and
- **Explain** what is better about the lifted version. This should be be done through contrasting how planning would proceed in the two alternative (lifted and non-lifted) structures and showing how the lifted structure has an advantage.

Note again: The plan fragment must be *sufficiently complete and informative*, which implies that you should include all details that help explain the usefulness of lifting.

(3 points)

4 Planning with Incomplete Information

In one of the later lectures, we discussed a number of different planning problems involving incomplete information about the world, including the Stochastic Shortest Path Problem (SSPP).

a) The Stochastic Shortest Path Problem is different from the general MDP (Markov Decision Process) problem. Nevertheless, an SSPP problem instance can be solved by a general MDP solver.

How do we transform the SSPP problem before giving it to an MDP solver, so that the solver cannot generate a policy that requires executing an infinite number of useful actions? How does this transformation guarantee that any MDP solution corresponds to an SSPP solution that achieves the SSPP-specific objectives?

(Part of this transformation may consist of turning costs into negative rewards, but this is not the part we are interested in here.)

(2 points)

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 every location must be 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 $\text{at}(t_1, t_1)$ 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 follows:

- 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*)