Försättsblad till skriftlig tentamen vid Linköpings Universitet

(fylls i av ansvarig)

Datum för tentamen	*2010-05-25*
Sal	*VALMAT*
Tid	*14-18*
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	*Martin Magnusson*
Telefon under skrivtid	
Besöker salen ca kl.	
Kursadministratör (namn + tfnnr + mailadress)	*Anna Grabska Eklund Ankn. 23 62, annek@ida.liu.se*
Tillåtna hjälpmedel	—
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	*Linjerat*
Antal exemplar i påsen	

1 Classical Plans (4 points)

We know the following:

- A classical plan π for a given planning problem instance is a sequence $[a_1, \ldots, a_n]$ of actions.
- A solution plan π for a given planning problem instance is *redundant* if there is a proper subsequence of π that is also a solution for $\mathcal{P}$. In other words, π is redundant if 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.
- A solution plan π for a given planning problem instance is *minimal* if there is no other solution for $\mathcal{P}$ 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 clearly why the solution is redundant – in other words, show which actions can be removed from the solution. **(2 points)**

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. Show clearly why the solution is neither redundant nor minimal. **(2 points)**

2 State-Based Heuristics (6 points)

We have discussed the h_m family of domain-independent admissible heuristics based on delete relaxation. You will now have to apply your knowledge about these heuristics to the standard logistics domain, as defined in Appendix A. You may assume a classical representation where the preconditions of an operator consist of a conjunction of positive atoms.

You should:

a) Explain the basic principle of *delete relaxation*. First, how is it applied to a planning domain – what actual changes are made in the domain? Second, how does this affect the search space? In other words, how does the search space *after* applying delete relaxation differ from the search space *before* delete relaxation was applied? **(2 points)**

b) Explain the intuitive meaning of $h_1(s)$, where s is a state. **(2 points)**

c) Generalize: Explain the intuitive meaning of $h_m(s)$ for $m \geq 1$. **(1 point)**

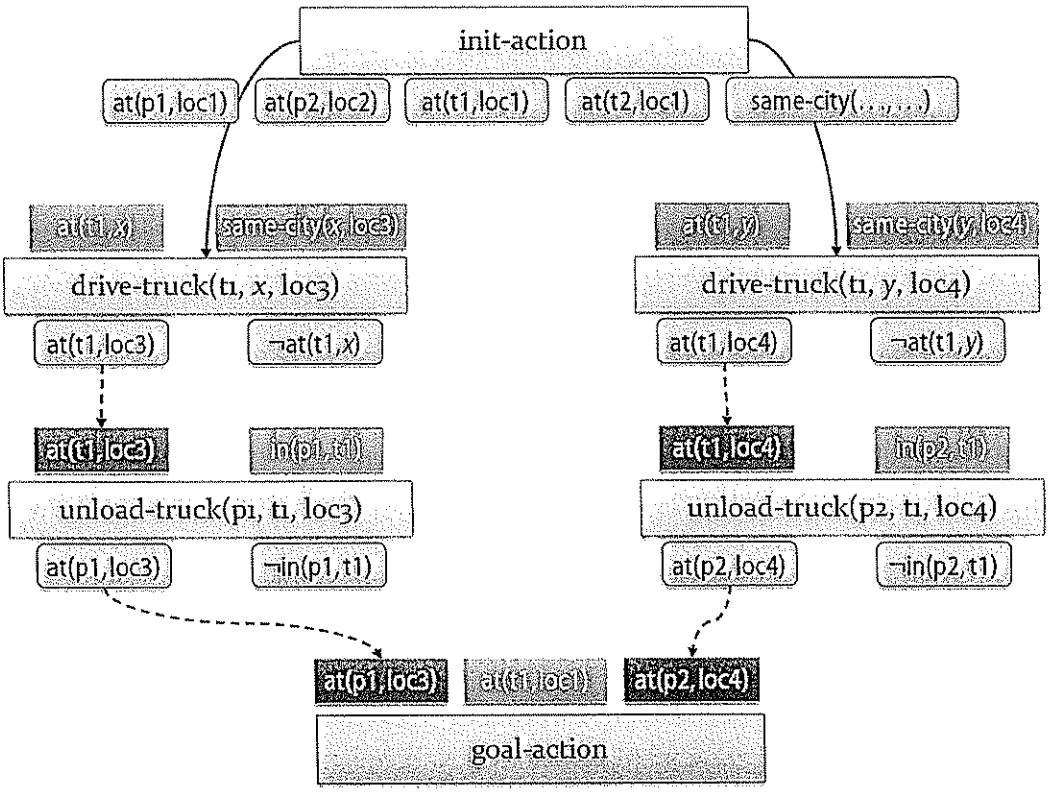
d) In general, heuristic functions are often intended to estimate the number of actions required to reach the goal from a certain state. However, the estimate given by the h_m family of heuristics is often far lower than the true number of remaining actions: $h_2(s)$ might return the value 20 for a state s even if reaching the goal from that state would require 1000 actions. *Why is the heuristic still useful?* In other words, how can domain-independent search methods used in planning still make use of these heuristic values to quickly find a path to the goal? **(1 point)**

3 Partial-Order Planning (6 points)

The figure below shows a partial plan generated by the PSP (Plan-Space Planning) procedure in the book, for the standard **logistics domain** defined in further detail in Appendix A. The problem instance used for this figure is quite small, using only the following objects:

- package: p1, p2
- truck: t1, t2
- airplane: (none)
- location: loc1, loc2, loc3, loc4
- airport: (none)

In the figure, effects are shown below each action, preconditions are shown above each action, and causal links are indicated by dashed arrows. Precedence constraints between actions are indicated by solid arrows and are not shown when there are also causal links between the same actions.



a) List all *threats* in this figure. Show exactly what is threatening what. **(2 points)**

b) For each of these threats, show at least one way in which it can be resolved. **(2 points)**

c) Show *all* ways in which you can resolve the *open goal* at(t1,y) belonging to the action drive-truck(t1,y,loc4) at the right hand side of the figure. You are *not* expected to resolve any new flaws that this may lead to. **(2 points)**

4 Hierarchical Task Networks (4p)

We continue by defining an HTN (hierarchical task network) formulation of the logistics domain. The assumption is that a logistics goal is 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. In other words, one possible initial network might be the following:

⟨deliver(package1,loc1,loc2), deliver(package2,loc1,loc3), deliver(package3,loc3,loc5)⟩

You should:

- Specify a set of *non-primitive* tasks and methods that can be used to decompose deliver(x, from, to) into subtasks, in a way that you eventually reach the predefined primitive tasks.

  We assume a TFD-like procedure is used (Total-order Forward Decomposition), with totally ordered methods and no means of interleaving 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.

  Since we use totally ordered methods, you are allowed to create inefficient solution plans in the specific sense that trucks and airplanes only move a single package at a time. However, solution plans must not include actions (primitive tasks) that "do nothing", such as driving from a location to the same location. **(4 points)**

Clarifications:

- 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 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
- package
- 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 not "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.
- location(x), airport(x), truck(x), airplane(x), vehicle(x) – type predicates.

We can then define the operators more formally as shown on the following page:

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