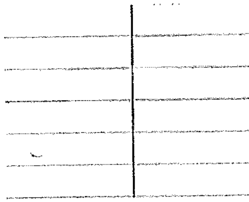
Försättsblad till skriftlig tentamen vid Linköpings universitet

Datum för tentamen	2019-06-05
Sal (1)	T2(16)
Tid	14-18
Utb. kod	TDDD48
Modul	TEN1
Utb. kodnamn/benämning Modulnamn/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 klockan	ca 15:00
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 2019-06-05

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*, `?location` – in italics or prefixed by "?"
- **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. An atom is also a formula – an "atomic (indivisible) formula" that does not contain smaller formulas.
- **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

In this question we will use the Sokoban domain and problem instance defined in PDDL in Appendix A. Please familiarize yourself with these before continuing.

Note: We are discussing the problem instance **defined in PDDL**, *not* the graphical illustration at the beginning of the appendix!

a) How many *states* exist for the given Sokoban problem instance, including all states, both reachable from the initial state and unreachable from the initial state?

   Show step by step how to calculate this number given a PDDL problem instance, with sufficient clarity that we can use your instructions to also compute the number of states for any other classical planning problem defined in PDDL. **(2 points)**

b) In the given Sokoban problem instance, is it true that for all states s reachable from the initial state, if you reach state s through some sequence and action and then leave s, you can then (always) return to s again?

   If so, motivate clearly. If not, provide a clearly comprehensible counterexample. You do not have to provide a full formal analysis based on the exact operator definitions; using your intutions about the Sokoban game is sufficient. **(1 point)**

c) Show the three first levels of the forward state space search tree for the given Sokoban problem instance, where the first level is the initial node and the other two levels are generated through applying applicable actions.

   For each state in the search tree that you illustrate, it is sufficient to specify (1) which `at` facts are *true*, and (2) the *value* of `total-cost`. You do not have to include the unchanging `IS-GOAL`, `IS-NONGOAL` or `MOVE-DIR` facts. You also do not have to include the `clear` facts. **(2 points)**

2 Partial Order Planning

In this question we will use the Gripper domain and problem instance defined in PDDL in Appendix B. Please familiarize yourself with these before continuing.

You will be required to **demonstrate a number of partially ordered plans.** 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 draw the plans 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.

Simplifying diagrams: If any aspect of a plan structure is not explicitly included in your diagram, you must explicitly and unambiguously state *how to infer what is missing* ("if there is an X with these properties, the plan actually also contains a Y with these other properties, but this is omitted from the diagram").

Questions:

a) Show the *initial partial plan* π_0 generated for the given Gripper problem instance by a typical partial-order causal link 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 ("initial node", "root node") generated in the search space. Think carefully about what the *very first* node is! **(2 points)**

b) Describe all *flaws* in the initial partial plan, and indicate the type of flaw. **(1 point)**

c) 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 subtask you do not have to illustrate each successor plan graphically. Instead, you can choose to explain clearly in writing how each successor would be constructed as a modification of π_0. However, you still need to indicate *all* changes that would be made relative to the initial partial plan, including any new constraints and relations as discussed above!

   You may use lifted successors or ground successors. **(2 points)**

3 Heuristics

a) Given the Gripper problem instance from Appendix B: Show at least 4 distinct ground *fact landmarks* that are *not already true* in the initial state.

The landmarks should use at least 2 different predicates. **(2 points)**

b) A state transition system, with its three components $\langle S, A, \gamma \rangle$, can also be seen as a mathematical *graph* consisting of *nodes* and labeled *edges*, allowing it to be more easily illustrated as a diagram.

When a *relaxation* is applied to a planning problem, how can this affect the mathematical graph?

In this question we are not interested in the exact definitions of specific relaxation techniques such as delete relaxation. We are also not interested in modifications at the level of the planning problem (such as how action definitions could change). Instead, we are asking how the *nodes and edges* could be affected by relaxations in a general way (addition, removal, other changes).

Describe at least two distinct types of change that can occur. Movitate why such changes would actually result in *relaxations*. **(2 points)**

c) Define the *optimal delete relaxation heuristic*. In other words, describe clearly how an algorithm would compute the heuristic value $h^{+}(s)$ given a planning domain, a problem instance and a state s.

You may *exemplify* using the Gripper domain from Appendix B if this helps you describe the heuristic more clearly. However, you must still describe a general computation procedure that works well for arbitrary domains. **(2 points)**

4 Planning with Incomplete Information

a) Explain the meaning of three types of planning: *Fully observable, partially observable* and *non-observable*. Include sufficient detail for the reader to understand the differences between these types of planning and the consequences for the planner and plan executor.

   Hints: This is about observations, but (1) *who* would observe something? (2) *What* can or can't this entity observe? (3) *When* would (or wouldn't) such observations be made? (4) When could the observations be *useful* and what would they be used for? (5) What information do you have if you *don't* have observations? **(2 points)**

b) Please define the Stochastic Shortest Path Problem (SSPP), by answering the following questions.

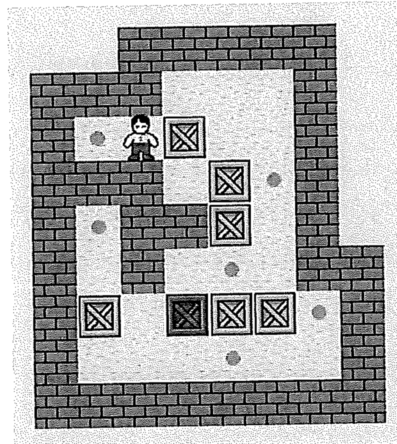
   - To define a classical planning problem instance, we can use a restricted form of state transition system $\langle S, A, \gamma \rangle$. A similar form of system can be defined for the Stochastic Shortest Path Problem. What is the main difference in this system, in terms of the actual components of the system? What is no longer there, what replaces it, ...?
   - A solution to the classical planning problem is a sequential plan. How can you specify a solution to the Stochastic Shortest Path Problem? Clearly explain the *typical* solution structure commonly used for both such problems and for other planning problems that are not fully deterministic.
   - In the classical planning problem, the *objective* is to find a sequential plan that starts in a known initial state s_0 and reaches a state belonging to a known set of goal states S_g, often while minimizing the cost of this solution.

     We described two possible objectives for the Stochastic Shortest Path Problem. Describe one of them!

   (3 points)

A The Sokoban Domain

Sokoban is a video game played on a board of squares, where the player moves crates (boxes) around in a warehouse in order to get them to a set of intended storage locations. The intended locations are not specific to a box: It does not matter which box is at which storage location. The player can move horizontally or vertically one step at a time, but cannot move through walls. The player can also push crates horizontally or vertically, also not through walls.



Illustrating example. This is *not* the same as the problem instance shown in PDDL below!

This domain can be modeled in PDDL as follows. Note that predicate symbols in ALL CAPS are fixed (not modified by any operator).

```
(define (domain sokoban-sequential)
  (:requirements :typing :action-costs)
  (:types thing location direction - object
          player stone - thing)
  (:predicates (clear ?l - location)
               (at ?t - thing ?l - location)
               (at-goal ?s - stone)
               (IS-GOAL ?l - location)
               (IS-NONGOAL ?l - location)
               (MOVE-DIR ?from ?to - location ?dir - direction))
  (:functions (total-cost) - number)

  (:action move
   :parameters (?p - player ?from ?to - location ?dir - direction)
   :precondition (and (at ?p ?from)
                      (clear ?to)
                      (MOVE-DIR ?from ?to ?dir)
                      )
   :effect       (and (not (at ?p ?from))
                      (not (clear ?to))
                      (at ?p ?to)
                      (clear ?from)
                      )
```

```
 )

(:action push-to-nongoal
 :parameters (?p - player ?s - stone
              ?ppos ?from ?to - location
              ?dir - direction)
 :precondition (and (at ?p ?ppos)
                    (at ?s ?from)
                    (clear ?to)
                    (MOVE-DIR ?ppos ?from ?dir)
                    (MOVE-DIR ?from ?to ?dir)
                    (IS-NONGOAL ?to)
                    )
 :effect       (and (not (at ?p ?ppos))
                    (not (at ?s ?from))
                    (not (clear ?to))
                    (at ?p ?from)
                    (at ?s ?to)
                    (clear ?ppos)
                    (not (at-goal ?s))
                    (increase (total-cost) 1)
                    )
 )

(:action push-to-goal
 :parameters (?p - player ?s - stone
              ?ppos ?from ?to - location
              ?dir - direction)
 :precondition (and (at ?p ?ppos)
                    (at ?s ?from)
                    (clear ?to)
                    (MOVE-DIR ?ppos ?from ?dir)
                    (MOVE-DIR ?from ?to ?dir)
                    (IS-GOAL ?to)
                    )
 :effect       (and (not (at ?p ?ppos))
                    (not (at ?s ?from))
                    (not (clear ?to))
                    (at ?p ?from)
                    (at ?s ?to)
                    (clear ?ppos)
                    (at-goal ?s)
                    (increase (total-cost) 1)
                    )
 )
)
```

A small Sokoban problem instance with 4 directions, 1 player and 7*7=49 locations can be defined as follows.

```
;; # is a wall #
;; $ is a box $
;; @ is a player @
;; . is a destination .

;; # #####
;;   #   #
;; ###$$@#
;; #   ###
;; #     #
;; # . . #
;; #######

(define (problem p024-microban-sequential)
  (:domain sokoban-sequential)
  (:objects
    dir-down - direction
    dir-left - direction
    dir-right - direction
    dir-up - direction
    player-01 - player
    pos-1-1 - location
    pos-1-2 - location
    pos-1-3 - location
    pos-1-4 - location
    pos-1-5 - location
    pos-1-6 - location
    pos-1-7 - location
    pos-2-1 - location
    pos-2-2 - location
    pos-2-3 - location
    pos-2-4 - location
    pos-2-5 - location
    pos-2-6 - location
    pos-2-7 - location
    pos-3-1 - location
    pos-3-2 - location
    pos-3-3 - location
    pos-3-4 - location
    pos-3-5 - location
    pos-3-6 - location
    pos-3-7 - location
    pos-4-1 - location
    pos-4-2 - location
    pos-4-3 - location
    pos-4-4 - location
    pos-4-5 - location
    pos-4-6 - location
    pos-4-7 - location
    pos-5-1 - location
    pos-5-2 - location
    pos-5-3 - location
    pos-5-4 - location
    pos-5-5 - location
    pos-5-6 - location
    pos-5-7 - location
    pos-6-1 - location
    pos-6-2 - location
    pos-6-3 - location
    pos-6-4 - location
    pos-6-5 - location
    pos-6-6 - location
    pos-6-7 - location
    pos-7-1 - location
    pos-7-2 - location
    pos-7-3 - location
    pos-7-4 - location
    pos-7-5 - location
    pos-7-6 - location
    pos-7-7 - location
    stone-01 - stone
    stone-02 - stone
  )
  (:init
    (IS-GOAL pos-3-6)
    (IS-GOAL pos-5-6)
    (IS-NONGOAL pos-1-1)
    (IS-NONGOAL pos-1-2)
    (IS-NONGOAL pos-1-3)
    (IS-NONGOAL pos-1-4)
    (IS-NONGOAL pos-1-5)
    (IS-NONGOAL pos-1-6)
    (IS-NONGOAL pos-1-7)
    (IS-NONGOAL pos-2-1)
    (IS-NONGOAL pos-2-2)
    (IS-NONGOAL pos-2-3)
    (IS-NONGOAL pos-2-4)
    (IS-NONGOAL pos-2-5)
    (IS-NONGOAL pos-2-6)
    (IS-NONGOAL pos-2-7)
    (IS-NONGOAL pos-3-1)
    (IS-NONGOAL pos-3-2)
    (IS-NONGOAL pos-3-3)
    (IS-NONGOAL pos-3-4)
    (IS-NONGOAL pos-3-5)
    (IS-NONGOAL pos-3-7)
```

```
(IS-NONGOAL pos-4-1)
(IS-NONGOAL pos-4-2)
(IS-NONGOAL pos-4-3)
(IS-NONGOAL pos-4-4)
(IS-NONGOAL pos-4-5)
(IS-NONGOAL pos-4-6)
(IS-NONGOAL pos-4-7)
(IS-NONGOAL pos-5-1)
(IS-NONGOAL pos-5-2)
(IS-NONGOAL pos-5-3)
(IS-NONGOAL pos-5-4)
(IS-NONGOAL pos-5-5)
(IS-NONGOAL pos-5-7)
(IS-NONGOAL pos-6-1)
(IS-NONGOAL pos-6-2)
(IS-NONGOAL pos-6-3)
(IS-NONGOAL pos-6-4)
(IS-NONGOAL pos-6-5)
(IS-NONGOAL pos-6-6)
(IS-NONGOAL pos-6-7)
(IS-NONGOAL pos-7-1)
(IS-NONGOAL pos-7-2)
(IS-NONGOAL pos-7-3)
(IS-NONGOAL pos-7-4)
(IS-NONGOAL pos-7-5)
(IS-NONGOAL pos-7-6)
(IS-NONGOAL pos-7-7)
(MOVE-DIR pos-1-2 pos-2-2 dir-right)
(MOVE-DIR pos-2-1 pos-2-2 dir-down)
(MOVE-DIR pos-2-2 pos-1-2 dir-left)
(MOVE-DIR pos-2-2 pos-2-1 dir-up)
(MOVE-DIR pos-2-4 pos-2-5 dir-down)
(MOVE-DIR pos-2-4 pos-3-4 dir-right)
(MOVE-DIR pos-2-5 pos-2-4 dir-up)
(MOVE-DIR pos-2-5 pos-2-6 dir-down)
(MOVE-DIR pos-2-5 pos-3-5 dir-right)
(MOVE-DIR pos-2-6 pos-2-5 dir-up)
(MOVE-DIR pos-2-6 pos-3-6 dir-right)
(MOVE-DIR pos-3-4 pos-2-4 dir-left)
(MOVE-DIR pos-3-4 pos-3-5 dir-down)
(MOVE-DIR pos-3-4 pos-4-4 dir-right)
(MOVE-DIR pos-3-5 pos-2-5 dir-left)
(MOVE-DIR pos-3-5 pos-3-4 dir-up)
(MOVE-DIR pos-3-5 pos-3-6 dir-down)
(MOVE-DIR pos-3-5 pos-4-5 dir-right)
(MOVE-DIR pos-3-6 pos-2-6 dir-left)
(MOVE-DIR pos-3-6 pos-3-5 dir-up)
(MOVE-DIR pos-3-6 pos-4-6 dir-right)
(MOVE-DIR pos-4-2 pos-4-3 dir-down)
(MOVE-DIR pos-4-2 pos-5-2 dir-right)
(MOVE-DIR pos-4-3 pos-4-2 dir-up)
(MOVE-DIR pos-4-3 pos-4-4 dir-down)
(MOVE-DIR pos-4-3 pos-5-3 dir-right)
(MOVE-DIR pos-4-4 pos-3-4 dir-left)
(MOVE-DIR pos-4-4 pos-4-3 dir-up)
(MOVE-DIR pos-4-4 pos-4-5 dir-down)
(MOVE-DIR pos-4-5 pos-3-5 dir-left)
(MOVE-DIR pos-4-5 pos-4-4 dir-up)
(MOVE-DIR pos-4-5 pos-4-6 dir-down)
(MOVE-DIR pos-4-5 pos-5-5 dir-right)
(MOVE-DIR pos-4-6 pos-3-6 dir-left)
(MOVE-DIR pos-4-6 pos-4-5 dir-up)
(MOVE-DIR pos-4-6 pos-5-6 dir-right)
(MOVE-DIR pos-5-2 pos-4-2 dir-left)
(MOVE-DIR pos-5-2 pos-5-3 dir-down)
(MOVE-DIR pos-5-2 pos-6-2 dir-right)
(MOVE-DIR pos-5-3 pos-4-3 dir-left)
(MOVE-DIR pos-5-3 pos-5-2 dir-up)
(MOVE-DIR pos-5-3 pos-6-3 dir-right)
(MOVE-DIR pos-5-5 pos-4-5 dir-left)
(MOVE-DIR pos-5-5 pos-5-6 dir-down)
(MOVE-DIR pos-5-5 pos-6-5 dir-right)
(MOVE-DIR pos-5-6 pos-4-6 dir-left)
(MOVE-DIR pos-5-6 pos-5-5 dir-up)
(MOVE-DIR pos-5-6 pos-6-6 dir-right)
(MOVE-DIR pos-6-2 pos-5-2 dir-left)
(MOVE-DIR pos-6-2 pos-6-3 dir-down)
(MOVE-DIR pos-6-3 pos-5-3 dir-left)
(MOVE-DIR pos-6-3 pos-6-2 dir-up)
(MOVE-DIR pos-6-5 pos-5-5 dir-left)
(MOVE-DIR pos-6-5 pos-6-6 dir-down)
(MOVE-DIR pos-6-6 pos-5-6 dir-left)
(MOVE-DIR pos-6-6 pos-6-5 dir-up)
(at player-01 pos-6-3)
(at stone-01 pos-4-3)
(at stone-02 pos-5-3)
(clear pos-1-2)
(clear pos-2-1)
(clear pos-2-2)
(clear pos-2-4)
(clear pos-2-5)
(clear pos-2-6)
(clear pos-3-4)
(clear pos-3-5)
(clear pos-3-6)
(clear pos-4-2)
(clear pos-4-4)
(clear pos-4-5)
(clear pos-4-6)
(clear pos-5-2)
(clear pos-5-5)
(clear pos-5-6)
```

```
    (clear pos-6-2)
    (clear pos-6-5)
    (clear pos-6-6)
    (= (total-cost) 0)
  )
  (:goal (and
    (at-goal stone-01)
    (at-goal stone-02)
  ))
  (:metric minimize (total-cost))
)
```

B The Gripper Domain

In the **Gripper** domain, a single robot carries balls between two rooms. The robot has two grippers and can therefore carry up to two balls at the same time. The goal is typically that all balls should be in the other room.

This domain is intentionally designed to demonstrate *redundancy*: There are n balls that can be picked up in $n!$ different orders, and each ball can be picked up in different grippers. None of these choices actually *matter*, so an optimal plan could easily be generated by a simple loop that moves the balls in an arbitrary order, for example in alphabetical order. The question is, does a particular planner or heuristic handle this redundancy efficiently, or does it spend a large amount of time investigating different "irrelevant" action orders?

```
(define (domain gripper-strips)
 (:predicates (room ?r) (ball ?b) (gripper ?g)
              (at-robby ?r)
              (at ?b ?r)
              (free ?g)
              (carry ?o ?g))

 (:action move
     :parameters  (?from ?to)
     :precondition (and  (room ?from) (room ?to) (at-robby ?from))
     :effect (and  (at-robby ?to)
                   (not (at-robby ?from))))

 (:action pick
     :parameters (?obj ?room ?gripper)
     :precondition  (and  (ball ?obj) (room ?room) (gripper ?gripper)
                          (at ?obj ?room) (at-robby ?room) (free ?gripper))
     :effect (and (carry ?obj ?gripper)
                  (not (at ?obj ?room))
                  (not (free ?gripper))))

 (:action drop
     :parameters  (?obj  ?room ?gripper)
     :precondition  (and  (ball ?obj) (room ?room) (gripper ?gripper)
                          (carry ?obj ?gripper) (at-robby ?room))
     :effect (and (at ?obj ?room)
                  (free ?gripper)
                  (not (carry ?obj ?gripper)))))
```

The following is a small problem instance with 4 balls.

```
(define (problem strips-gripper-x-1)
 (:domain gripper-strips)
 (:objects rooma roomb ball4 ball3 ball2 ball1 left right)
 (:init (room rooma)
        (room roomb)
        (ball ball4)
        (ball ball3)
        (ball ball2)
        (ball ball1)
        (at-robby rooma)
        (free left)
        (free right)
        (at ball4 rooma)
        (at ball3 rooma)
        (at ball2 rooma)
        (at ball1 rooma)
        (gripper left)
        (gripper right))
 (:goal (and (at ball4 roomb)
             (at ball3 roomb)
             (at ball2 roomb)
             (at ball1 roomb))))
```